# TIBCO SmartSockets™

## cxxipc Class Library

*Software Release 6.8*
*July 2006*

## Important Information

SOME TIBCO SOFTWARE EMBEDS OR BUNDLES OTHER TIBCO SOFTWARE. USE OF SUCH EMBEDDED OR BUNDLED TIBCO SOFTWARE IS SOLELY TO ENABLE THE FUNCTIONALITY (OR PROVIDE LIMITED ADD-ON FUNCTIONALITY) OF THE LICENSED TIBCO SOFTWARE. THE EMBEDDED OR BUNDLED SOFTWARE IS NOT LICENSED TO BE USED OR ACCESSED BY ANY OTHER TIBCO SOFTWARE OR FOR ANY OTHER PURPOSE.

USE OF TIBCO SOFTWARE AND THIS DOCUMENT IS SUBJECT TO THE TERMS AND CONDITIONS OF A LICENSE AGREEMENT FOUND IN EITHER A SEPARATELY EXECUTED SOFTWARE LICENSE AGREEMENT, OR, IF THERE IS NO SUCH SEPARATE AGREEMENT, THE CLICKWRAP END USER LICENSE AGREEMENT WHICH IS DISPLAYED DURING DOWNLOAD OR INSTALLATION OF THE SOFTWARE (AND WHICH IS DUPLICATED IN THE *TIBCO SMARTSOCKETS INSTALLATION GUIDE*). USE OF THIS DOCUMENT IS SUBJECT TO THOSE TERMS AND CONDITIONS, AND YOUR USE HEREOF SHALL CONSTITUTE ACCEPTANCE OF AND AN AGREEMENT TO BE BOUND BY THE SAME.



THIS DOCUMENT IS PROVIDED “AS IS” WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THIS DOCUMENT. TIBCO SOFTWARE INC. MAY MAKE IMPROVEMENTS AND / OR CHANGES IN THE PRODUCT(S) AND / OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

# Contents

**Figures** . . . . . **vii**

**Preface** . . . . . **ix**

Intended Audience . . . . . x

Related Documentation . . . . . xi

TIBCO Product Documentation . . . . . xi

Using the Online Documentation . . . . . xi

Conventions Used in This Manual . . . . . xi

Typeface Conventions . . . . . xii

Notational Conventions . . . . . xiii

Identifiers . . . . . xiii

Case . . . . . xiv

How to Contact TIBCO Customer Support . . . . . xiv

**Chapter 1 Overview** . . . . . **1**

The cxxipc Library . . . . . 2

C++ Class Organization . . . . . 2

Relation to C Language API . . . . . 3

Inheritance Hierarchy . . . . . 5

Mapping the C API to C++ Class Member Functions . . . . . 7

When to Use the C++ Class Library . . . . . 8

**Chapter 2 Using the C++ Class Library** . . . . . **9**

Example: C++ TIBCO SmartSockets Application . . . . . 10

Sender Program . . . . . 10

Receiver Program . . . . . 12

Compiling, Linking, and Running . . . . . 13

Error Handling . . . . . 16

Error Checking and Static Member Functions . . . . . 18

Include Files . . . . . 18

Source File Distribution . . . . . 19

Using Threads . . . . . 21

**Chapter 3 Messages . . . . . 23**

TipcMsg Class . . . . . 24
Vacant and Non-Vacant TipcMsg Objects . . . . . 25
TipcMt Class . . . . . 26
Standard Message Types . . . . . 26
User-Defined Message Types . . . . . 26
Vacant and Non-Vacant TipcMt Objects . . . . . 27
Working With Messages . . . . . 27
TipcMsg Construction . . . . . 27
Appending and Accessing TipcMsg Data Fields . . . . . 30
Running Message Programs . . . . . 33
Using Array Fields . . . . . 34
TipcMsg and T_IPC_MSG Reference Counts . . . . . 37
Message Files . . . . . 38
Using Message Files . . . . . 38

**Chapter 4 Connections . . . . . 41**

Class Hierarchy Of Connections . . . . . 42
Constructors . . . . . 42
Working with Connections . . . . . 43
Server Source Code . . . . . 44
Client Source Code . . . . . 50
Running Server and Client Programs . . . . . 52
Creating Connections . . . . . 54
Destroying a Connection . . . . . 56
Callbacks . . . . . 57

**Chapter 5 Publish and Subscribe . . . . . 59**

Overview . . . . . 60
RTserver and RTclient Composition . . . . . 61
The TipcSrv Class . . . . . 62
Working With The TipcSrv Class . . . . . 63
Common Header Source Code . . . . . 64
Receiver Side . . . . . 64
Sender Side . . . . . 67

**Chapter 6 C++ Class Reference . . . . . 71**

Class Overview . . . . . 73
Relation to the C API . . . . . 74
Naming Conventions . . . . . 74
Data Structures . . . . . 75
Include Files . . . . . 76
Utility Functions and Macros . . . . . 77
Compiling and Linking . . . . . 77
On UNIX . . . . . 77
On OpenVMS . . . . . 78
On Windows . . . . . 79
Error Handling . . . . . 79
C++ Class Descriptions . . . . . 80
TipcConn . . . . . 81
Construction/Destruction . . . . . 81
Protected Data . . . . . 82
Type Conversion Operators . . . . . 82
Member Functions . . . . . 83
Static Public Member Functions . . . . . 93
Example . . . . . 93
TipcConnClient . . . . . 95
Construction/Destruction . . . . . 95
Example . . . . . 96
TipcConnServer . . . . . 97
Construction/Destruction . . . . . 97
Public Member Function . . . . . 98
Example . . . . . 98
TipcMon . . . . . 99
Construction/Destruction . . . . . 99
Public Member Functions . . . . . 100
TipcMonClient . . . . . 104
Construction/Destruction . . . . . 104
Public Member Functions . . . . . 104
TipcMonServer . . . . . 108
Construction/Destruction . . . . . 108
Public Member Functions . . . . . 108

TipcMsg . . . 110
Construction/Destruction . . . 111
Public Member Functions . . . 112
Conversion Operators . . . 145
Operators . . . 145
Manipulators . . . 151
Parameterized Manipulators . . . 151
Static Public Member Functions . . . 152
Protected Member Functions . . . 155
Example . . . 156
TipcMsgFile . . . 158
Construction/Destruction . . . 158
Operators . . . 159
Example . . . 159
TipcMt . . . 161
Construction/Destruction . . . 161
Public Member Functions . . . 163
Conversion Operators . . . 166
Static Public Member Functions . . . 167
Example . . . 167
TipcSrv . . . 168
Public Member Functions . . . 169
Static Public Member Functions . . . 181
Example . . . 184
Tobj . . . 185
Protected Construction/Destruction . . . 185
Public Member Functions . . . 185
Operators . . . 186
Example . . . 186

**Chapter 7 Project Monitoring . . . 187**

Available Monitoring Information . . . 188
Hierarchy Of Monitoring Classes . . . 190
Working With the Monitoring Classes . . . 190
Code Example . . . 191
Running a Monitoring Program . . . 193

**Index . . . 195**

# Figures

Figure 1 A Programmer's View of the SmartSockets API . . . . . 3
Figure 2 A Programmer's View of the SmartSockets C++ Classes . . . . . 4
Figure 3 Inheritance Hierarchy . . . . . 5
Figure 4 TipcConn Class Inheritance Hierarchy . . . . . 42
Figure 5 RTserver and RTclient Architecture . . . . . 61
Figure 6 TipcConn Class Inheritance Hierarchy . . . . . 62
Figure 7 Monitoring Inheritance Hierarchy . . . . . 190

# Preface

TIBCO SmartSockets is a message-oriented middleware product that enables programs to communicate quickly, reliably, and securely across:

- local area networks (LANs)
- wide area networks (WANs)
- the Internet

TIBCO SmartSockets takes care of network interfaces, guarantees delivery of messages, handles communications protocols, and directs recovery after system or network problems. This enables you to focus on higher-level requirements rather than the underlying complexities of the network.

This guide describes the `cxxipc` library, the C++ class library for the SmartSockets inter-process communication (IPC) library. The `cxxipc` library is included with TIBCO SmartSockets, Version 6.2 and higher, for backwards compatibility. No new features or functions added to TIBCO SmartSockets, Version 6.2 and higher, are supported with the `cxxipc` library. For all new SmartSockets C++ development, use the `sscpp` library, described in the *TIBCO SmartSockets C++ User's Guide*.

The TIBCO SmartSockets cxxipc Class Library provides:

- an overview of the `cxxipc` library organization
- a description of its relation to the C language APIs upon which it is based
- detailed descriptions and examples of how to set up, compile and run C++ applications using the `cxxipc` library

## Topics


- *Related Documentation, page xi*
- *Conventions Used in This Manual, page xi*
- *How to Contact TIBCO Customer Support, page xiv*

# Intended Audience

This guide is intended for C++ programmers who plan to use SmartSockets in an object-oriented manner without using the C API.

Some prerequisite knowledge is needed to understand the concepts and examples in this guide:

- working knowledge of C++
- familiarity with the operating system is required for developing SmartSockets applications (UNIX, Windows, OpenVMS, or whatever platform is running SmartSockets). This includes knowing how to log in, log out, edit a text file, change directories, list files, and compile, link, and run a program.
- understand general messaging and publish/subscribe concepts and terminology
- familiarity with the SmartSockets messaging concepts covered in the *TIBCO SmartSockets User's Guide.*

# Related Documentation

This section lists documentation resources you may find useful.

## TIBCO Product Documentation

The following documents form the SmartSockets documentation set:

- *TIBCO SmartSockets API Quick Reference*
- *TIBCO SmartSockets Application Programming Interface*
- *TIBCO SmartSockets C++ User's Guide*
- *TIBCO SmartSockets cxxipc Class Library*
- *TIBCO SmartSockets Installation Guide*
- *TIBCO SmartSockets Java Library User's Guide and Tutorial*
- *TIBCO SmartSockets .NET User's Guide and Tutorial*
- *TIBCO SmartSockets Tutorial*
- *TIBCO SmartSockets User's Guide*
- *TIBCO SmartSockets Utilities*
- *TIBCO SmartSockets C++ and Java Class Libraries*

  C++ class library and Java application programming interface (API) reference materials are available in HTML format only. Access the references through the TIBCO HTML documentation interface.

## Using the Online Documentation

The SmartSockets documentation files are available for you to download separately, or you can request a copy of the TIBCO Documentation CD.

# Conventions Used in This Manual

This manual uses the following conventions.

## Typeface Conventions

This manual uses the following typeface conventions

| Example | Use |
|---|---|
| `monospace` | This monospace font is used for program output and code example listing and for file names, commands, configuration file parameters, and literal programming elements in running text. |
| **`monospace bold`** | This bold monospace font indicates characters in a command line that you must type exactly as shown. This font is also used for emphasis in code examples. |
| *Italic* | Italic text is used as follows:<br>• In code examples, file names etc., for text that should be replaced with an actual value. For example: "Select *install-dir* / `runexample.bat`."<br>• For document titles.<br>• For emphasis. |
| **Bold** | Bold text indicates actions you take when using a GUI, for example, click **OK**, or choose **Edit** from the menu. It is intended to help you skim through procedures when you are familiar with them and just want a reminder.<br>Submenus and options of a menu item are indicated with an angle bracket, for example, **Menu > Submenu.** |
|  | Warning. The accompanying text describes a condition that severely affects the functioning of the software. |
|  | Note. Be sure you read the accompanying text for important information. |
|  | Tip. The accompanying text may be especially helpful. |

## Notational Conventions

The notational conventions in the table below are used for describing command syntax. When used in this context, do not type the brackets listed in the table as part of a command line.

| Notation | Description | Use |
|---|---|---|
| [ ] | Brackets | Used to enclose an optional item in the command syntax. |
| < > | Angle Brackets | Used to enclose a name (usually in *Italic*) that represents an argument for which you substitute a value when you use the command. This convention is not used for XML or HTML examples or other situations where the angle brackets are part of the code. |
| { } | Curly Brackets | Used to enclose two or more items among which you can choose only one at a time.<br>Vertical bars (\|) separate the choices within the curly brackets. |
| ... | Ellipsis | Indicates that you can repeat an item any number of times in the command line. |

## Identifiers

The term identifier is used to refer to a valid character string that names entities created in a SmartSockets application. The string starts with an underscore (_) or alphabetic character and is followed by zero or more letters, digits, percent signs (%), or underscores. No other special characters are valid. The maximum length of the string is 63 characters. Identifiers are not case-sensitive.

These are examples of valid identifiers:

```
EPS
battery_11
K11

__
_all
```

These are invalid identifiers:

```
20
battery-11
@com
$amount
```

### Case

Function names are case-sensitive, and must use the mixed-case format you see in the text. For example, TipcMsgCreate, TipcSrvStop, and TipcMonClientMsgTrafficPoll are SmartSockets functions and must use the case as shown.

Monitoring messages are also case-sensitive, and should be all upper case, such as T_MT_MON_SERVER_NAMES_POLL_CALL. This makes it easy to distinguish them from option or function names.

Although option names are not case-sensitive, they are usually presented in text with mixed case, to help distinguish them from commands or other items. For example, Server_Names, Unique_Subject, and Project are all SmartSockets options.

Identifiers used with the products in the SmartSockets family are not case-sensitive. For example, the identifiers `thermal` and `THERMAL` are equivalent in all processes.

In UNIX, shell commands and filenames are case-sensitive, though they might not be in other operating systems, such as Windows. To make it easier to port applications between operating systems, always specify filenames in lower case.

## How to Contact TIBCO Customer Support

For comments or problems with this manual or the software it addresses, please contact TIBCO Support Services as follows.

- For an overview of TIBCO Support Services, and information about getting started with TIBCO Product Support, visit this site:

  http://www.tibco.com/services/support/default.jsp

- If you already have a valid maintenance or support contract, visit this site:

  http://support.tibco.com

  Entry to this site requires a username and password. If you do not have a username, you can request one.

# Chapter 1 Overview

This chapter gives an overview of the features of the TIBCO SmartSockets `cxxipc` library and shows how these features are accessed. It is assumed throughout this guide that you are familiar with the C language-based SmartSockets model and the C++ programming language. For more background information on the SmartSockets IPC model, see the *TIBCO SmartSockets User's Guide* and the *TIBCO SmartSockets Application Programming Interface*.

TIBCO SmartSockets now includes a new C++ class library, the `sscpp` library. The `sscpp` library is the preferred library for new development. For more information on the features and use of the `sscpp` library, see the *TIBCO SmartSockets C++ User's Guide*.

The TIBCO SmartSockets cxxipc Class Library describes the `cxxipc` library, which is included with TIBCO SmartSockets, Version 6.2 and higher, for backwards compatibility only. It does not support any new features or functions added to later releases of SmartSockets. All new development should use the `sscpp` library.

## Topics


- *C++ Class Organization, page 2*
- *When to Use the C++ Class Library, page 8*

## The cxxipc Library

The TIBCO SmartSockets `cxxipc` class library provides access to interprocess communication (IPC) facilities from the C++ programming language. Key features of this class library include:

- organization of the (C language-based) TIBCO SmartSockets Application Programming Interface (API) into C++ classes and associated member functions
- a C++ class inheritance hierarchy further organizing the C language SmartSockets API functions into an object-oriented framework suitable for derivation and modification
- overloaded insertion and extraction operators that simplify much of the syntax of assembling, sending, receiving, and gaining visibility into SmartSockets messages
- introduction of virtual and polymorphic member functions to shorten the naming and simplify the organization of the C language SmartSockets API functions upon which the SmartSockets C++ class library is based
- inclusion of example C++ programs that show how common C language SmartSockets application programs are expressed in C++ using the SmartSockets C++ class library

SmartSockets now also includes a new C++ class library, the `sscpp` library. The `sscpp` library is preferred for new SmartSockets C++ development. It supports multiple connections and additional utilities functions from the C language SmartSockets API, and includes improved callback and error handling. For more information, see the *TIBCO SmartSockets C++ User's Guide*.

## C++ Class Organization

You use the C++ class library by constructing C++ objects provided by the class library and invoking their member functions. Also, the provided classes may be used as base classes for your classes that modify or extend the functionality of the SmartSockets C++ class library.

## Relation to C Language API

The SmartSockets C++ class library is implemented as a layered product whose internals make calls to the C language API. Although this C++ class library provides all the functionality of the underlying C language API at the C++ level, the features of the conventional C language API remain fully available if you wish to call the C language API directly. Figure 1 and Figure 2 relate the conventional C language SmartSockets API to particular C++ classes provided in the SmartSockets C++ class library:

*Figure 1 A Programmer's View of the SmartSockets API*

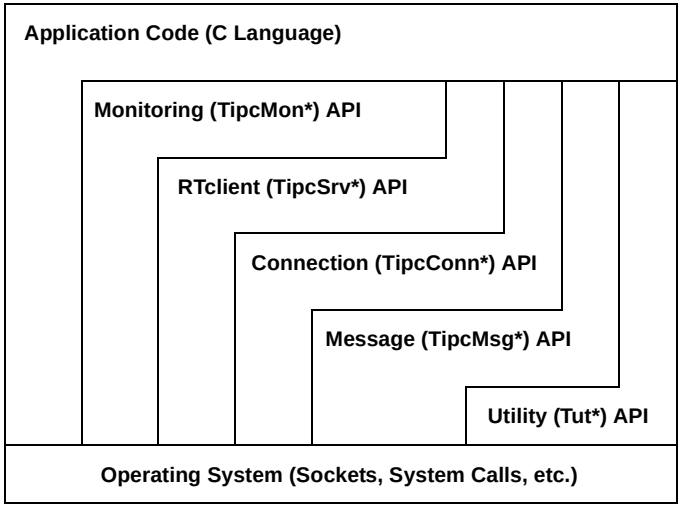
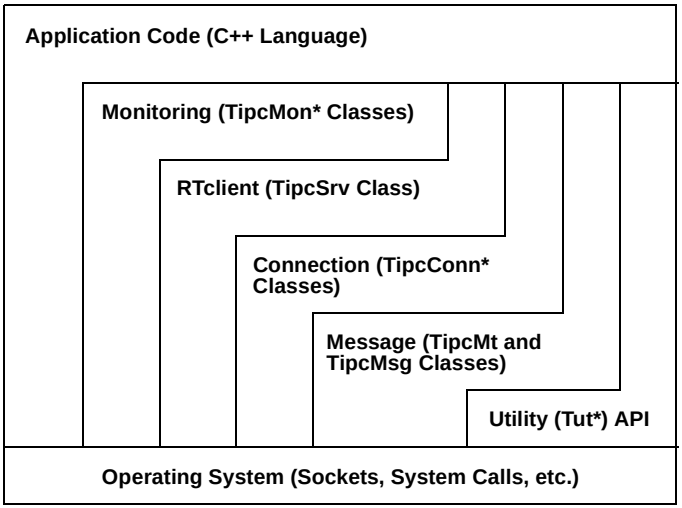


*Figure 2 A Programmer's View of the SmartSockets C++ Classes*



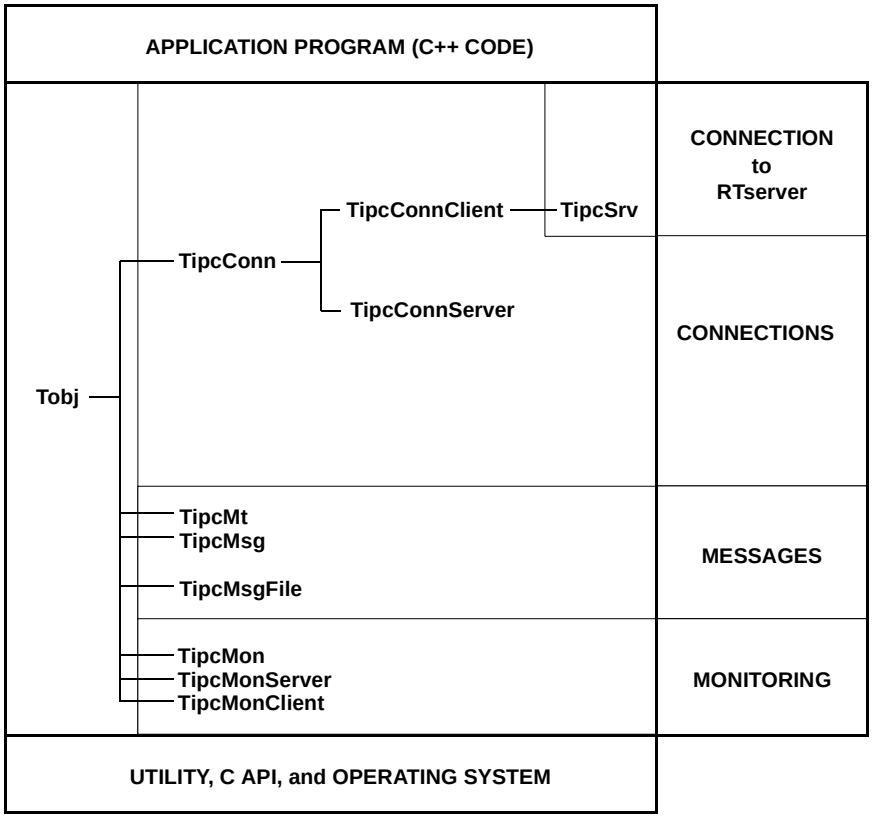

As indicated in Figure 2, some of the C language API of SmartSockets, more specifically the utility (Tut*) calls, are not wrapped in the SmartSockets C++ class library. Therefore, you must call these C functions directly from C++ when required. All other C language API calls (including, for example, all the TipcConn*, TipcSrv*, TipcMon*, TipcMsg*, and TipcMt* calls) are available using either the provided C++ wrapper methods from the C++ class library or through direct calls to the C language API.

It is recommended that you try to consistently use either the C++ class library or C API, not both.

## Inheritance Hierarchy

The SmartSockets C++ class library includes a number of classes organized into an inheritance hierarchy. These classes are either directly or indirectly derived from the Tobj base class, as shown in Figure 3.

*Figure 3 Inheritance Hierarchy*




The organization of the SmartSockets class hierarchy reflects and clarifies several object-oriented aspects of SmartSockets that are already present in its C language implementation.

### Clear Identification of System Objects

In the C API, the fundamental objects of interest are messages, message types, peer-to-peer connections, RTclient functionality, and monitoring. These system objects are naturally expressed by the SmartSockets classes TipcMsg, TipcMt, TipcConn, TipcSrv, and TipcMon.

### Grouping of Functions by Similarity

In the C API, functions are grouped by C language prefixes to indicate similarities in functionality. For example, the functions TipcMtLookup and TipcMtTraverse share the TipcMt prefix. Such similarity is expressed in C++ using member functions. The corresponding C++ class library methods are TipcMt::Lookup and TipcMt::Traverse.

Likewise, consider the SmartSockets functions TipcSrvGetNumQueued and TipcSrvGetSocket. These two functions are represented in the C++ class library by the TipcSrv class member functions TipcSrv::NumQueued() and TipcSrv::Socket().

In C, the corresponding `set` method to TipcSrvGetSocket is called TipcSrvSetSocket. In C++, the `set` method TipcSrv::Socket is differentiated from its `get` variant by its requiring an argument. The corresponding `set` C++ prototype is TipcSrv::Socket(`T_INT4` *socket_arg*).

### Implied Derivation of Functionality

The C language functionality of RTclient (the TipcSrv* API) expands the capabilities of purely peer-to-peer, connection-based message passing (the TipcConn* API) by adding a publish-subscribe message routing capability.

In C++, the augmentation and possible modification of a set of base capabilities is expressed using inheritance and virtual member functions. In the SmartSockets C++ class library, the functionality that SmartSockets connections and RTserver connections have in common is first expressed in the TipcConn base class using virtual functions that the TipcSrv derived class then reimplements.

One example of such common functionality reimplemented in the derived class is represented by the functions TipcConn::Timeout() and TipcSrv::Timeout(). The former member function implementation calls the C function TipcConnGetTimeout(), while the latter reimplements the method, calling TipcSrvGetTimeout() instead.

### Functional Abstraction

In C++, an abstract class is a class used only as a base class of some other class; no objects of an abstract class may be created except as objects of a class from which it is derived. Abstract classes are used to express the notion of a general concept (such as shape) of which only concrete variants (such as rectangle or triangle) can actually be used. One such SmartSockets abstract base class is the Tobj class, which manages the error handling conventions that many Tipc* C API functions have in common.

## Mapping the C API to C++ Class Member Functions

The *TIBCO SmartSockets Application Programming Interface* describes in detail the functionality of each of the C API functions. Generally speaking, each such C language function corresponds to a particular member function of a SmartSockets C++ class whose identity can be determined from Table 1.

*Table 1 Relationship of C Language API to C++ Classes*

| For the C Language API Prefix | Functionality Found in the C++ Class |
|---|---|
| TipcCb* | (Callback only; not wrapped) |
| TipcConn* | TipcConn |
| TipcConnCreate* | TipcConn<br>TipcConnServer<br>TipcConnClient |
| TipcMon* | TipcMon |
| TipcMonClient* | TipcMonClient |
| TipcMonServer* | TipcMonServer |
| TipcMsg* | TipcMsg |
| TipcMt* | TipcMt |
| TipcSrv* | TipcConn<br>TipcSrv |

### How to Find Class Member Function Wrappers of the C API

The index of this guide contains an entry for most C API functions that are wrapped by a C++ class. The C API functions are listed in the index under the entry C API. Go to the page listed for the C API function you are looking up. That page contains information about the C++ member function that wraps that particular C API function.

C++ classes wrapping C API functions that support multiple connections are included with the `sscpp` library. For more information, see the *TIBCO SmartSockets C++ User's Guide*.

# When to Use the C++ Class Library

The SmartSockets C++ class library organizes the C language API into an object-oriented C++ class library. Therefore, if you are already working in C++ and with object-oriented software in general, you reap the greatest benefits in using the SmartSockets C++ class library. Because the SmartSockets C++ class library adds no additional IPC capabilities beyond what is already present in the SmartSockets C language API, the choice whether to use the SmartSockets C++ class library or the C language API is essentially up to you.

Some of the key advantages of the SmartSockets C++ class library are:

- Less complicated API

  By grouping together related methods into a particular C++ class, the SmartSockets C++ class library reduces the name length and the numbers of arguments of many of the corresponding C API functions, leading to shorter, more readable code.

- Simplified memory management

  In C++, the allocation, initialization, and deallocation of objects is often localized to class constructors and destructors. Where possible, the classes included in the SmartSockets C++ class library use this technique to assume responsibility for the memory allocation and destruction of SmartSockets system objects on your behalf. You are then freed from having to explicitly allocate objects using functions such as TipcMsgCreate, and from having to explicitly deallocate objects using functions such as TipcMsgDestroy.

- Reusability and object-orientation

  Several of the provided C++ classes in the SmartSockets C++ class library are available to you for reuse through inheritance and re-implementation of their virtual member functions. Also, you may syntactically associate callback functions with SmartSockets C++ classes by deriving your own classes from SmartSockets C++ classes, such as TipcConnClient. Using this technique syntactically groups the callback functionality of a network connection with the connection itself, inside a particular C++ class, making the resulting code more suitable for extension, reuse, and modification.

- Simpler encoding and decoding of messages

  The SmartSockets TipcMsg class, described in the later chapters of this guide, describes how C++ insertion and extraction operator overloading is used in the TipcMsg class to unify much of the process of constructing, reading, and writing the messages inside a SmartSockets application program.

# Chapter 2 Using the C++ Class Library

This chapter introduces a simple C++ TIBCO SmartSockets application using the C++ class library. The mechanics of compiling and linking applications using the class library are also included.

## Topics


- *Example: C++ TIBCO SmartSockets Application, page 10*
- *Error Handling, page 16*
- *Include Files, page 18*
- *Source File Distribution, page 19*

# Example: C++ TIBCO SmartSockets Application

These two C++ language example programs illustrate a simple SmartSockets application that uses the C++ class library. The first program, called sender, is a program that uses RTserver to publish a message consisting of two strings to the second program, called receiver.

## Sender Program

This source listing is a program written in C++ that uses the RTserver to publish two strings in the message to another program. A discussion of the highlights follows the program example.

```
#include <rtworks/cxxipc.hxx>

int main(int argc, char **argv)
{
    // Set the name of the project
    T_OPTION option = TutOptionLookup("project");
    TutOptionSetEnum(option, "ipc_example");

    // Connect to RTserver
    TipcSrv& srv = TipcSrv::InstanceCreate(T_IPC_SRV_CONN_FULL);

    // Create a message
    TipcMsg msg(T_MT_STRING_DATA);
    msg.Dest("/demo");

    // Build the message with two string fields: "x" and "Hello World"
    msg << "x" << "Hello World" << Check;
    if (!msg) {
      TutOut("Error appending fields of TipcMsg object\n");
      return T_EXIT_FAILURE;
    }

    // Publish the message
    srv.Send(msg);
    srv.Flush();
    srv.Destroy(T_IPC_SRV_CONN_NONE);  // force blocking close

    return T_EXIT_SUCCESS;
} // main
```

The first line of the program, `#include <rtworks/cxxipc.hxx>`, must be included in every program that uses the SmartSockets C++ class library. It contains a series of `#include`'s for the various header files of the class library and is provided as a convenience to the programmer.

Moving inside the body of the `main()` function, the SmartSockets utility functions TutOptionLookup and TutOptionSetEnum are used to set the project name for the sender program. The receiver program uses the same project name. See the *TIBCO SmartSockets User's Guide* for more information on project names.

After the project name is set, obtain a handle to a TipcSrv object and create a connection to RTserver. This is typically done with this line of code:

```
TipcSrv& srv = TipcSrv::InstanceCreate(T_IPC_SRV_CONN_FULL);
```

Note that a TipcSrv object is not created using a constructor; instead, a reference to a TipcSrv object is returned. The TipcSrv class is designed this way because a given RTclient is only allowed to have, at most, one connection to RTserver at a time. The class library manages the construction of the TipcSrv object to encapsulate this restriction. See TipcSrv on page 168 for more information about the TipcSrv class.

Following the call to TipcSrv::InstanceCreate(), a TipcMsg object is constructed by passing the T_MT_STRING_DATA message type as an argument to the constructor. The message destination is set to `/demo`. See the *TIBCO SmartSockets User's Guide* for more information on the destination property of messages.

The next line of code in sender appends data fields to the `TipcMsg` object. This example illustrates a means of appending data that is unique to the C++ class library as compared to the C API. Data is appended by using overloaded insertion operators in a function chain. The function chain employs the Check manipulator to set a status flag inside of the `TipcMsg` object if any error occurred during the function chain's execution. Data can also be appended using the overloaded TipcMsg::Append member function, which provides an interface similar to the C API TipcMsgAppend* functions. See Chapter 3, Messages, for more information on constructing messages.

The final step is to publish the message, which is accomplished by calling the TipcSrv member functions TipcSrv::Send(), TipcSrv::Flush(), and TipcSrv::Destroy(). See TipcSrv on page 168 for more information about the TipcSrv class.

## Receiver Program

This source listing is a program written in C++ that uses the RTserver to receive two strings in the message from another program. After the listing is a brief discussion of some program highlights.

```
#include <rtworks/cxxipc.hxx>

int main(int argc, char **argv)
{
  // Set the name of the project
  T_OPTION option = TutOptionLookup("project");
  TutOptionSetEnum(option, "ipc_example");

  // Connect to RTserver
  TipcSrv& srv = TipcSrv::InstanceCreate(T_IPC_SRV_CONN_FULL);

  // Subscribe to receive any messages published to the "/demo"
  // subject
  srv.SubjectSubscribe("/demo", TRUE);

  // Get the next incoming message
  TipcMsg msg(srv.Next(T_TIMEOUT_FOREVER));
  if (!srv) {
      TutOut("Error creating TipcMsg object\n");
      return T_EXIT_FAILURE;
  }
  // Set the pointer to first field in message
  msg.Current(0);

  T_STR var_name;
  T_STR var_value;

  // Extract the information from the received message
  msg >> var_name >> var_value >> Check;
  if (!msg) {
      TutOut("Error reading fields of TipcMsg object\n");
      return T_EXIT_FAILURE;
  }

  // Display the values on stdout
  TutOut("Variable Name  = %s\n", var_name);
  TutOut("Variable Value = %s\n", var_value);
  srv.Destroy(T_IPC_SRV_CONN_NONE); // force blocking close

  return T_EXIT_SUCCESS;
} // main
```

The first line of the program is `#include <rtworks/cxxipc.hxx>` just as it was in the sender program earlier. Inside the main function, the same C API calls are also used to set the project name option and create the connection to RTserver.

The receiver program next subscribes to the appropriate subject, `/demo`, by calling the TipcSrv::SubjectSubscribe member function. Note that the sender program designated `/demo` as the destination of the message. See the *TIBCO SmartSockets User's Guide* for information regarding subjects in RTclient.

At this point, the receiver waits for a message from RTserver. This is done by calling the TipcSrv::Next member function using T_TIMEOUT_FOREVER as the argument to the call. The result of the Next member function is either a T_IPC_MSG or a `NULL`. This value is passed to the TipcMsg constructor for the `msg` object. If the TipcMsg constructor is passed a `NULL`, then the internal status of the object is set to `FALSE`. The status of the `msg` object is queried with `operator!()`. See TipcMsg on page 110 for more details on the TipcMsg class.

After the receiver program receives a message from RTserver, the data fields are extracted from the message. The first step in the extraction is to set the current field of the message to the first field by using the TipcMsg::Current member function. Next, the actual data is extracted in a C++ function chain of overloaded extraction operators. As with the insertion function chain used in the sender to append data fields to a message, the extraction function chain employs the Check manipulator to set a status flag inside of the TipcMsg object if any error occurred during the function chain's execution. Data are also extracted using the overloaded TipcMsg::Next member functions, which provide an interface similar to the C API TipcMsgNext* functions. See Chapter 3, Messages, for more information on extracting data fields from TipcMsg objects.

Finally, the program outputs the values of the data fields by outputting the data using the TutOut function calls. Note that this example avoids using cout from IOSTREAM. The reason for this is that some SmartSockets functions use TutOut to display status information. Because TutOut uses the stdout stream and cout typically uses its own output stream, it is recommended that TutOut be used in RTclient programs, rather than cout, so that output is printed in sequential order.

## Compiling, Linking, and Running

To compile, link, and run the example programs, first you must either copy the programs to your own directory or have write permission in this directory:

**UNIX:**

```
$RTHOME/examples/cxxipc
```

**OpenVMS:**

```
RTHOME:[EXAMPLES.CXXIPC]
```

**Windows:**

```
%RTHOME%\examples\cxxipc
```

Use these commands to compile and link the programs:

**UNIX:**

```
$ rtlink -cxx -o sndr.x sndr.cxx
$ rtlink -cxx -o rcvr.x rcvr.cxx
```

**OpenVMS:**

```
$ cxx sndr.cxx
$ cxx rcvr.cxx
$ rtlink -cxx /exec=sndr.exe sndr.obj
$ rtlink -cxx /exec=rcvr.exe rcvr.obj
```

**Windows:**

```
$ nmake /f sndrw32m.mak
$ nmake /f rcvrw32m.mak
```

On a UNIX system the `rtlink` command by default uses the C compiler `cc` command to compile and link. Specifying the `-cxx` flag tells `rtlink` to use the native C++ compiler (for example, on a Solaris platform the compiler is named `CC`, on AIX `xlC`, on Digital UNIX `cxx`, and so on) and adds the SmartSockets C++ class library to the list of libraries to be linked into the executable. To use a C++ compiler other than the default compiler, set the environment variable `CC` to the name of the compiler. `rtlink` then uses this compiler.

For example, these commands are used to compile and link on UNIX with the GNU C++ compiler `g++`:

**UNIX:**

```
$ env CC=g++ rtlink -cxx -o sndr.x sndr.cxx
$ env CC=g++ rtlink -cxx -o rcvr.x rcvr.cxx
```

Start RTserver before you run the program. Use this command to start RTserver:

```
$ rtserver
```

On platforms that support both 32- and 64-bit, use the `rtserver64` command to run the 64-bit version of the rtserver script.

To run the programs, start the receiving process first in one terminal emulator window and then the sending process in another terminal emulator window.

Start up the receiving program in the first window:

**UNIX:**

```
$ rcvr.x
```

**OpenVMS:**

```
$ run rcvr.exe
```

**Windows:**

```
$ rcvr.exe
```

Start up the sending program in the second window:

**UNIX:**

```
$ sndr.x
```

**OpenVMS:**

```
$ run sndr.exe
```

**Windows:**

```
$ sndr.exe
```

The output from the receiving process is similar to this:

```
Connecting to project <ipc_example> on <_node> RTserver.
Using local protocol.
Message from RTserver: Connection established.
Start subscribing to subject </_workstation1_6605>.
Variable Name  = x
Variable Value = Hello World
```

The output from the sending process is similar to this:

```
Connecting to project <ipc_example> on <_node> RTserver.
Using local protocol.
Message from RTserver: Connection established.
Start subscribing to subject </_workstation1_6607>.
```

## Error Handling

Most of the SmartSockets functions return FALSE on failure, and TRUE on success. The corresponding C++ member functions instead set a status flag in the object. A block written in C looks similar to this:

```
if (!TipcMsgAppendStr(msg, "voltage")) {
    TutOut("Could not append first field.\n");
    return T_EXIT_FAILURE;
}
```

In C++, this same program looks similar to this:

```
msg << "voltage";
if (!msg) {
    TutOut("Could not append first field.\n");
    return T_EXIT_FAILURE;
}
```

The question arises, what does the ! operator do when applied to a C++ object such as msg? The answer is contained in the declaration of the Tobj class at the root of the C++ class library hierarchy, as in this example:

```
#include <rtworks/ipc.h>

class Tobj {
protected:
    T_BOOL    _status;

    Tobj();
    virtual Tobj() {}

public:
    //----------------------------------------
    //  Methods for accessing status
    //----------------------------------------
    T_BOOL operator!() const { return !_status; }
    T_BOOL Status() const { return _status; }
};
```

As shown above, the Tobj class manages a status flag, which is set to TRUE at construction. Each object derived from Tobj inherits its own private version of this status flag. You gain access to the status flag value by calling either the overloaded operator!() or the Status() member function. In this way, Tobj permits value return function calls without precluding the ability to error-check each call to the underlying C API. To see how this is accomplished, consider TipcMsg::Dest():

First, TipcMsg::Dest() calls the C API's TipcMsgGetDest(). Then it sets the status flag of TipcMsg to the status value returned by TipcMsgGetDest(). Finally, it returns the value of the destination field of the message as the member function's return value. This is illustrated in this example by using T_BOOL TipcMsg::Dest().

```
// ================================================================
//..TipcMsg::Dest -- get the destination property of a message
T_STR TipcMsg::Dest()
{
  T_STR dest_return = "";   // initialize variable

  _status = TipcMsgGetDest(_msg, &dest_return);
  return dest_return;
}
```

When using TipcMsg::Dest(), you access the status by using either the `!` operator or the Status() member function. For example:

```
TipcMsg msg;

T_STR the_dest = msg.Dest();

// one way to access the status
if (!msg) {
  // handle the error
}

// another way to access the status
if (!msg.Status()) {
  // handle the error
}
```

Typically, C++ class library member functions that access information return a value. Otherwise, class library member functions return the status and set the internal status flag. This is illustrated by T_BOOL TipcMsg::Dest(T_STR dest):

```
// ================================================================
//..TipcMsg::Dest -- set the destination property of a message
T_BOOL TipcMsg::Dest(T_STR dest)
{
  _status = TipcMsgSetDest(_msg, dest);
  return _status;
}
```

The use of the C API's status return convention can be avoided by using Tobj::operator!() or Tobj::Status(). The only exception to this is when using static member functions. See Error Checking and Static Member Functions.

Because the Tobj constructor is protected, the class is rendered into an abstract base class. Users of the C++ class library interact with derived classes of Tobj only.

### Error Checking and Static Member Functions

Because a static member function call involves no C++ this pointer, there is also no corresponding Tobj base class instance in which to store a result status should a static member function fail. Therefore, to perform accurate error checking after static member function calls, consult the global SmartSockets error number.

## Include Files

Code written in C++ that uses the C++ class library must include the header file `cxxipc.hxx`, which is located this directory:

**UNIX:**

```
$RTHOME/include/$RTARCH/rtworks
```

**OpenVMS:**

```
RTHOME:[INCLUDE.RTWORKS]
```

**Windows:**

```
%RTHOME%\include\rtworks
```

This include file automatically includes all the header files used for interprocess communication using the C++ class library.

# Source File Distribution

The C++ class library is distributed in binary and source code format. The binaries are compiled with a native compiler from the vendor of a particular platform. (For instance, a Sun SPARCompiler was used to compile binaries for Solaris.) Because C++ compilers may not necessarily generate binaries that are compatible with other C++ compilers, the C++ sources are also distributed so that you can compile the class library with a C++ compiler compatible with your environment. The source files are located in this directory:

**UNIX:**

```
$RTHOME/source/cxxipc
```

**OpenVMS:**

```
RTHOME:[SOURCE.CXXIPC]
```

**Windows:**

```
%RTHOME%\source\cxxipc
```

A sample makefile is included with the source files to build the C++ library. The library name produced and the name of the directory are:

**UNIX:**

```
$RTHOME/lib/$RTARCH/libcxxipc.a
```

**OpenVMS:**

```
RTHOME:[LIB.'RTARCH]cxxipc.olb
```

**Windows:**

```
%RTHOME%\lib\%RTARCH%\ipcx.lib
```

On UNIX, the sample makefile builds the library with the GNU C++ compiler `g++`.

This makefile does not overwrite this TIBCO-supplied library from the product distribution:

**UNIX:**
`libcxxipc.a`

**OpenVMS:**
`rtcxxipc.olb`

**Windows:**
`tipcx.lib`

Each source file (`.cxx`) in the source code has a corresponding header file (`.hxx`) that contains the declaration of a SmartSockets class, in this directory:

**UNIX:**
`$RTHOME/include/$RTARCH`

**OpenVMS**:
`RTHOME:[INCLUDE.RTWORKS]`

**Windows:**
`%RTHOME%\include\rtworks`

The source file organization is:

| Source File | SmartSockets C++ Class Implementation |
|---|---|
| `tconn.cxx` | TipcConn<br>TipcConnServer<br>TipcConnClient |
| `tmon.cxx` | TipcMon |
| `tmonclt.cxx` | TipcMonClient |
| `tmonsrv.cxx` | TipcMonServer |
| `tmsg.cxx` | TipcMsg |
| `tmsgfile.cxx` | TipcMsgFile |
| `tmsgmanp.cxx` | TipcMsgManip |

| Source File | SmartSockets C++ Class Implementation |
|---|---|
| `tmt.cxx` | TipcMt |
| `tobj.cxx` | Tobj |
| `tsrv.cxx` | TipcSrv |

## Using Threads

The SmartSockets threads API is not wrapped, because it is part of the utilities library (Tut*). This includes the function TipcInitThreads that must be called if your application uses threads. See the *TIBCO SmartSockets Utilities* for more details on the threads API.

# Chapter 3 Messages

Within a SmartSockets application, interprocess communication occurs through messages. A message is a packet of information sent from one process to one or more other processes providing instructions or data for the receiving process. Messages can carry many different kinds of information, including:

- variable data in the form of a series of variable names and their values (the most common use of a message for SmartSockets)
- commands to a process' command interface
- user-defined binary data, such as images or multi-byte strings
- monitoring information about RTserver and RTclient processes
- guaranteed message delivery (GMD) information about other messages

All of these different kinds of messages are classified by message types. For example, numeric variable data is typically sent in a NUMERIC_DATA type of message, and an operator warning is typically sent in a WARNING type of message. A SmartSockets application can use both the standard message types provided with SmartSockets, as well as user-defined message types.

This chapter describes message composition, message types, how to create message types, and how to work with messages using the C++ class library. This chapter should be considered a companion to the *TIBCO SmartSockets User's Guide*. You may wish to read the detailed information on SmartSockets messages in the *TIBCO SmartSockets User's Guide* before proceeding further in this chapter. The intent of this chapter is merely to focus on constructing and managing messages using TipcMsg and TipcMt classes.

## Topics


- *TipcMsg Class, page 24*
- *TipcMt Class, page 26*
- *Working With Messages, page 27*
- *Message Files, page 38*

# TipcMsg Class

The TipcMsg class is composed of several parts. These parts consist of a message (C type T_IPC_MSG), and various member functions that manipulate an object of the TipcMsg class.

In the C API, messages, message types, and their associated data are manipulated primarily through functions that begin with the TipcMsg* or TipcMt* prefix. In the C++ class library, the corresponding C++ methods are grouped primarily within two C++ classes called TipcMsg and TipcMt. A third class, called Tobj, manages a status flag used for error checking when manipulating messages and message types in the C++ class library.

You manipulate the type, sender, delivery mode, destination, priority, delivery timeout, load-balancing mode, header string encoding, user-defined field, and read-only properties of a message by calling methods of the TipcMsg class. The data part of a message can be manipulated in two ways, overloaded Append() and Next() member functions, or overloaded insertion and extraction operators (operator<<() and operator>>() respectively). The Append() and Next() member functions are C++ implementations of the C API's TipcMsgAppend* and TipcMsgNext* functions. The insertion and extraction operators provide an interface similar to IOSTREAM classes for appending and accessing data fields from a message.

It is recommended that the use of Append() and Next() member functions not be mixed with the insertion and extraction operators. The two methods represent different paradigms that may not work well together.

## Vacant and Non-Vacant TipcMsg Objects

A vacant TipcMsg object is one that is not currently managing a C type T_IPC_MSG message. Likewise, a non-vacant TipcMsg object is one that is currently managing a C type T_IPC_MSG message. The simple TipcMsg constructor (described in Using A Simple Constructor on page 28) can be used to explicitly create a vacant TipcMsg object. All other constructors create objects that manage a C type T_IPC_MSG. The major difference between a vacant and a non-vacant TipcMsg object is that data fields can only be inserted into and extracted from a non-vacant TipcMsg object.

Vacant TipcMsg objects also appear as a result of an operation on a non-vacant TipcMsg object. A vacant TipcMsg object is created whenever destroy responsibility is taken from the caller, or a member function fails and needs to return a valid TipcMsg object.

These member functions take destroy responsibility from the caller, and also take a non-vacant TipcMsg object and make it vacant:

- TipcConn::Insert()
- TipcMsg::Destroy()
- TipcSrv::Insert()

These member functions may return a vacant TipcMsg object when they fail:

- TipcConn::Next()
- TipcConn::Search()
- TipcConn::SearchType()
- TipcConn::SendRpc()
- TipcMsg::Clone()
- TipcMsg::Create()
- TipcMsgFile::operator>>()
- TipcSrv::Next()
- TipcSrv::Search()
- TipcSrv::SearchType()
- TipcSrv::SendRpc()

The TipcMsg::Vacant() member function can be used to determine if a TipcMsg object is vacant or non-vacant.

# TipcMt Class

The TipcMt class is a C++ interface to the TipcMt-prefix SmartSockets C API function calls. The TipcMt class contains a pointer to the message type (C type T_IPC_MT).

## Standard Message Types

A TipcMt object is created by specifying one of the standard message types to the TipcMt constructor. This is accomplished in one of four ways: specifying a C type T_IPC_MT variable, specifying a standard message type number, specifying a standard message type name, or specifying another TipcMt object. For example:

```
T_IPC_MT mt;

// Get a C type T_IPC_MT using a C API call.
mt = TipcMtLookup("info");

// Create TipcMt objects four different ways
TipcMt mt1(mt);
TipcMt mt2("info");
TipcMt mt3(T_MT_INFO);
TipcMt mt4(mt1);
```

In a similar manner, TipcMsg objects are also constructed by using a TipcMt object, message type number, a message type name, or an initialized C type T_IPC_MT variable. Depending on your needs, it is possible to save an additional step by constructing TipcMsg objects directly from a message type number, a message type name, or an initialized C type T_IPC_MT variable rather than from a constructed TipcMt object.

## User-Defined Message Types

An important use of the TipcMt class is creating user-defined message types when a standard message type does not meet your requirements. A user-defined message type is created by calling the appropriate constructor. The resulting TipcMt object is then used in a TipcMsg constructor to create a message of the user-defined type. For example:

```
#define XYZ_COORD_DATA 1001
// Create user defined message type.
TipcMt mt("xyz_coord_data", XYZ_COORD_DATA, "int4 int4 int4");
```

When a TipcMt object goes out of scope, the message type is not destroyed, just the TipcMt object it was managing. A TipcMt object is reconstructed using the user-defined data type. For example:

```
TipcMt mt("xyz_coord_data");
```

### Vacant and Non-Vacant TipcMt Objects

Like the TipcMsg class, the TipcMt class has vacant and non-vacant objects. A vacant TipcMt object is one that does not refer to a C T_IPC_MT variable. A non-vacant TipcMt object does refer to a C T_IPC_MT variable. A vacant object is constructed this way:

```
TipcMt mt_vacant;
```

A vacant TipcMt object is useful for creating and looking up global callbacks. See Global Callbacks on page 57 for more information on how to create a global callback.

The TipcMt::Destroy() member function leaves a TipcMt object vacant after it calls the C API function TipcMtDestroy(). The TipcMt::Create() member function leaves a TipcMt object vacant whenever it fails. The overloaded TipcMt::Lookup() member functions also leave an object vacant whenever a message type look up fails.

The TipcMt::Vacant() member function can be used to determine if a TipcMt object is vacant or non-vacant.

## Working With Messages

This section explains how the TipcMsg and TipcMt classes are used in a C++ program and provides example programs that show how the classes are used.

### TipcMsg Construction

There are several ways of constructing a TipcMsg object. The construction method to use depends on what information is available at the time the object is being constructed.

### Using A Simple Constructor

The simplest constructor of the TipcMsg class is:

```
TipcMsg::TipcMsg()
```

This constructor creates a vacant TipcMsg object that is later initialized with a C type T_IPC_MSG or a non-vacant TipcMsg object using operator=(). For example:

```
// Create a couple of vacant TipcMsg objects using the simple
// TipcMsg constructor
TipcMsg vacant_msg_obj_1, vacant_msg_obj_2;

// Initialize vacant object using operator=().
TipcMsg non_vacant_msg_obj(T_MT_TIME);
vacant_msg_obj_1 = non_vacant_msg_obj;

// Initialize vacant object using the Create member function.
vacant_msg_obj_2.Create(T_MT_INFO);
```

### Using A TipcMt Object

A TipcMsg object is created with a TipcMt object using the `TipcMsg::TipcMsg(const TipcMt& mt_obj)` constructor. For example:

```
const T_INT4 MY_MSG_TYPE = 100;

TipcMt mt_obj("my_msg_type", MY_MSG_TYPE, "int4 real4");

TipcMsg msg(mt_obj);
```

### Using The Convenience Constructor

A TipcMsg object is created using what is called the convenience constructor. For example:

```
TipcMsg::TipcMsg(T_INT4 mt_num,
                 T_STR destination = (T_STR)NULL,
                 T_STR sender = (T_STR)NULL);
```

This constructor is so named because it can construct a TipcMsg object without using a C type T_IPC_MT variable or a TipcMt object, thus saving the steps necessary in assigning the C type T_IPC_MT variable or instantiating a TipcMt object prior to the construction of the TipcMsg object. The constructor also initializes a few message properties. The only required argument is *mt_num*, the message type number. The other two parameters have default values and thus can be optionally supplied. The default values for destination and sender are `null` string pointers. For example:

```
// create a few TipcMsg objects.
TipcMsg msg1(T_MT_NUMERIC_DATA);
TipcMsg msg2(T_MT_INFO, "/stocks");
TipcMsg msg3(T_MT_CONTROL, "/system", "/admin");
```

### Using a Message Type Name

A TipcMsg object is created with a message type name using:

`TipcMsg::TipcMsg(`*mt_name*`)`

Like the convenience constructor above, this constructor can construct a TipcMsg object without using a C type T_IPC_MT variable or a TipcMt object. For example:

```
// create a TipcMsg object using a message type name
TipcMsg msg("info");
```

### Using The Copy Constructor

A TipcMsg object is created for another TipcMsg object using the copy constructor:

`TipcMsg::TipcMsg(`*msg_obj*`)`

This constructor is invoked when passing a TipcMsg by value to a function, when a TipcMsg object is returned by a function, or when an explicit copy is made. For example:

```
TipcMsg userCode(TipcMsg b)   // pass by value: copy main()'s a to b
{
    TipcMsg c(b);         // explicit copy: copy from b to c
    return c;             // return by value: copy from c to main()'s d
}
int main()
{
    TipcMsg a;
    TipcMsg d = userCode(a);
}
```

When this constructor is used, the message reference count increments by one.

### Using A C Type T_IPC_MSG

A TipcMsg object is created with a C type T_IPC_MSG variable using the constructor:

`TipcMsg::TipcMsg(T_IPC_MSG` *c_type_msg*`)`

For example:

```
T_IPC_MSG msg;

// create a C type T_IPC_MSG message
msg = TipcMsgCreate(TipcMtLookup("info"));

// construct a TipcMsg object using a C type T_IPC_MSG
TipcMsg msg_object(msg);
```

When this constructor is used, the message reference count increments by one.

## Appending and Accessing TipcMsg Data Fields

There are two means of appending and accessing data fields of non-vacant TipcMsg objects. The first way is to use overloaded TipcMsg::Append() and TipcMsg::Next() functions. These functions duplicate the C API TipcMsgAppend* and TipcMsgNext* functionality. An alternative means is to use the overloaded C++ insertion and extraction operators (operator<<() and operator>>(), respectively). To illustrate the two methods, a program is implemented twice in this example, using one of the two methods each time.

### Using Append and Next Functions

This example illustrates the usage of the Append() and Next() member functions of the TipcMsg class. These member functions are overloaded and accept many types of arguments. These functions duplicate the functionality of the TipcMsgAppend* and TipcMsgNext* function sets in the C API. See TipcMsg on page 110 for more information on these member functions.

The source code file for this example is located in this directory:

**UNIX:**
`$RTHOME/examples/cxxipc`

**OpenVMS:**
`RTHOME:[EXAMPLES.CXXIPC]`

**Windows:**
`%RTHOME%\examples\cxxipc`

In some cases, C++ literals must be cast to an appropriate data type so that the C++ compiler selects the correct overloaded Append member function. For example, `33.4534` could be interpreted as a `T_REAL4` or a `T_REAL8` by the compiler. To clear the ambiguity, use either `(T_REAL4)33.4534` or `(T_REAL8)33.4534`.

```
// msg1.cxx -- messages example using Append and Next
#include <rtworks/cxxipc.hxx>

int main()
{
  T_STR str_val;
  T_REAL8 real8_val;

  TutOut("Create the message.\n");
  TipcMsg msg(T_MT_NUMERIC_DATA, "/_saratoga", "/thermal");
  if (!msg) {
    TutOut("Could not create message.\n");
    return T_EXIT_FAILURE;
  }
```

```
  TutOut("Append fields.\n");
  if (!msg.Append("voltage")) {
    TutOut("Could not append first field.\n");
    return T_EXIT_FAILURE;
  }

 // A cast to T_REAL8 here is to ensure that the C++ compiler
 // selects the correct overloaded Append member function.

if (!msg.Append((T_REAL8)33.4534)) {
    TutOut("Could not append second field.\n");
    return T_EXIT_FAILURE;
  }

  if (!msg.Append("switch_pos")) {
    TutOut("Could not append third field.\n");
    return T_EXIT_FAILURE;
  }

  if (!msg.Append((T_REAL8)0.0)) {
    TutOut("Could not append fourth field.\n");
    return T_EXIT_FAILURE;
  }

  TutOut("Access fields.\n");
  if (!msg.Current(0)) {
    TutOut("Could not set current field to first field.\n");
    return T_EXIT_FAILURE;
  }

  if (!msg.Next(&str_val)) {
    TutOut("Could not read first field.\n");
    return T_EXIT_FAILURE ;
  }

    if (!msg.Next(&real8_val)) {
    TutOut("Could not read second field.\n");
    return T_EXIT_FAILURE;
  }
  TutOut("%s = %s\n", str_val, TutRealToStr(real8_val));
  if (!msg.Next(&str_val)) {
    TutOut("Could not read third field.\n");
    return T_EXIT_FAILURE;
  }

  if (!msg.Next(&real8_val)) {
    TutOut("Could not read fourth field.\n");
    return T_EXIT_FAILURE;
  }
  TutOut("%s = %s\n", str_val, TutRealToStr(real8_val));

  // msg is destroyed when TipcMsg goes out of scope and
  // the TipcMsg destructor is called.
  return T_EXIT_SUCCESS;  // all done

}  // main
```

## Using Insertion and Extraction Operators

This example illustrates the usage of insertion and extraction operators (operator<<() and operator>>(), respectively) of the TipcMsg class. These operators are overloaded and accept many types of arguments. These operators are an alternative means of appending and accessing the data fields of a message, and can be used instead of the Append() and Next() member functions. Please refer to TipcMsg on page 110 for more information on these operators.

The source code file for this example is located in this directory:

**UNIX:**

```
$RTHOME/examples/cxxipc
```

**OpenVMS:**

```
RTHOME:[EXAMPLES.CXXIPC]
```

**Windows:**

```
%RTHOME%\examples\cxxipc
```

As with the Append() member functions, some literals must be cast to an appropriate data type so that the C++ compiler selects the correct overloaded insertion operator. This example shows how this was applied for a T_REAL8 cast of `33.4534`.

```
// msg2.cxx -- messages example using insertion
// and extraction operators
#include <rtworks/cxxipc.hxx>

int main()
{
  T_STR str_val;
  T_REAL8 real8_val;

  TutOut("Create the message.\n");
  TipcMsg msg(T_MT_NUMERIC_DATA, "/_saratoga_5415", "/thermal");
  if (!msg) {
    TutOut("Could not create message.\n");
    return T_EXIT_FAILURE;
  }

  TutOut("Append fields.\n");

 // A cast to T_REAL8 here is to ensure that the C++ compiler
 // selects the correct overloaded insertion operator.
  msg << "voltage" << (T_REAL8)33.4534
      << "switch_pos" << (T_REAL8)0.0 << Check;
  if (!msg) {
    TutOut("Could not append data fields to message\n");
    return T_EXIT_FAILURE;
  }
```

```
  TutOut("Access fields.\n");
  if (!msg.Current(0)) {
    TutOut("Could not set current field to first field.\n");
    return T_EXIT_FAILURE;
  }
  msg >> str_val >> real8_val >> Check;
  if (!msg) {
    TutOut("Could not read data fields from message.\n");
    return T_EXIT_FAILURE;
  }
  TutOut("%s = %s\n", str_val, TutRealToStr(real8_val));

  msg >> str_val >> real8_val >> Check;
  if (!msg) {
    TutOut("Could not read data fields from message.\n");
    return T_EXIT_FAILURE;
  }
  TutOut("%s = %s\n", str_val, TutRealToStr(real8_val));

 // msg is destroyed when TipcMsg goes out of scope and
 // the TipcMsg destructor is called.
  return T_EXIT_SUCCESS;  // all done

}  // main
```

## Running Message Programs

Compile and link `msg1` and `msg2` (see Compiling, Linking, and Running on page 13 for generic instructions). Use these commands to run the programs:

**UNIX:**

```
$ msg1.x
$ msg2.x
```

**OpenVMS:**

```
$ run msg1.exe
$ run msg2.exe
```

**Windows:**

```
$ msg1.exe
$ msg2.exe
```

The output from the `msg1` and `msg2` process is similar to this output:

```
Create the message.
Append fields.
Access fields.
voltage = 33.4534
switch_pos = 0
```

## Using Array Fields

Array fields are appended and accessed in two ways just like scalar data types. One way is to use the overloaded Append() and Next() member functions of the TipcMsg class. Like the C API, the Append() member function requires the number of elements in the array as a parameter. For example:

```
T_REAL8 real8_array[10]; // can hold up to 10 values

real8_array[0] = 3.0;
real8_array[1] = 4.0;
real8_array[2] = 55.75;
if (!msg.Append(real8_array, 3)) {
    // error
}
```

Array fields can be accessed using the Next() member function. Like the C API, Next() requires a parameter where it stores the number of elements in the array. For example:

```
T_REAL8 *real8_array;
T_INT4 array_size;

if (!msg.Next(&real8_array, &array_size)) {
    // error
}
```

The second way of appending and accessing array fields is using the insertion and extraction operators. When using these operators, the SetSize() and GetSize() manipulators are employed to set and retrieve, respectively, the number of elements in the array field. To append an array, use SetSize() and operator<<(). For example:

```
T_REAL8 real8_array[10]; // can hold up to 10 values

real8_array[0] = 3.0;
real8_array[1] = 4.0;
real8_array[2] = 55.75;

msg << SetSize(3) << real8_array << Check;
if (!msg) {
    // error
}
```

Note that the Check manipulator is used in the function chain to set the error status of the TipcMsg object, `msg`. To access an array, use operator>>() and GetSize(). For example:

```
T_REAL8 *real8_array;
T_INT4 array_size;

msg >> real8_array >> GetSize(&array_size) >> Check;
if (!msg) {
    // error
}
```

The order in which the SetSize() and GetSize() manipulators are called is critical. When inserting an array, the SetSize() manipulator must be used prior to inserting the array. When extracting an array, the GetSize() manipulator must be used after extracting the array. If the manipulators are used improperly, unpredictable results may occur.

### Message Array Fields

A message may contain a field that is an array of messages. This example shows how a message array can be declared in C:

```
T_IPC_MSG msg_array[3];
```

A message array is inserted into the message with TipcMsgAppendMsgArray(), and it is extracted with TipcMsgNextMsgArray(). See the detailed descriptions in the *TIBCO SmartSockets Application Programming Interface* reference for examples of how these functions are used.

In C++, the equivalent of a T_IPC_MSG array is the TipcMsg array. An array of TipcMsg objects can be inserted into a message using either TipcMsg::Append() or the TipcMsg insertion operator. Likewise, a TipcMsg array can be extracted from a message using either TipcMsg::Next() or the TipcMsg extraction operator.

Inserting a TipcMsg array into a message in C++ is simple. For example:

```
TipcMsg msg1("T_MT_INFO");
TipcMsg msg2("T_MT_INFO");
TipcMsg msg_array[3];

msg_array[0].Create(T_MT_NUMERIC_DATA);
msg_array[1].Create(T_MT_NUMERIC_DATA);
msg_array[2].Create(T_MT_CONTROL);

// Use the insertion operator to insert an array
msg1 << SetSize(3) << msg_array << Check;
if (!msg1) {
    // error
}
// Use the Append member function to insert an array
msg2.Append(msg_array, 3);
if (!msg2) {
    // error
}
```

Both the insertion operator and the Append() member function take a TipcMsg array, convert it to a T_IPC_MSG array, and append the T_IPC_MSG array to the T_IPC_MSG message managed by a TipcMsg object.

Extracting a message is not as straightforward as inserting one for two reasons. First, the size of a message array is unknown until after it is extracted. Second, the T_IPC_MSG array extracted from a message must be converted into a TipcMsg array. Therefore, both the extraction operator and the Next() member function

extract a T_IPC_MSG array from a message, determine the size of the array, and then dynamically allocate a TipcMsg array to hold the elements of the extracted T_IPC_MSG array. The address of the allocated TipcMsg array is returned to the caller. The burden of deallocating a TipcMsg array is on the caller. This is an example of extracting a TipcMsg array from a message:

```
TipcConnClient conn("tcp:_node:5252");
TipcMsg msg;
TipcMsg *msg_array;
T_INT4 size;

msg = conn.Next(10.0);
if (!conn) {
    // error
}
// set the message to the first field
msg.Current(0);

// extract a message array using the extraction operator
msg >> &msg_array >> GetSize(&size) >> Check;
  if (!msg) {
    // error
}
// print the contents of the message array
for (T_INT4 i = 0; i < size; i++) {
    msg_array[i].Print(TutOut);
}

// delete the message array
delete[] msg_array;

// set the message to the first field
msg.Current(0);

// extract a message array using the Next member function
msg.Next(&msg_array, &size);
if (!msg) {
    // error
}

// print the contents of the message array
for (T_INT4 i = 0; i < size; i++) {
    msg_array[i].Print(TutOut);
}

// delete the message array
delete[] msg_array;
```

### Deleting Extracted TipcMsg Arrays

When a TipcMsg array is deleted or goes out of scope, the TipcMsg destructor is called on each element of the array. The destructor always passes the internal T_IPC_MSG variable managed by each TipcMsg array element to TipcMsgDestroy() unless the internal T_IPC_MSG is a read-only message. In other words, whenever a TipcMsg array is destroyed, the internal T_IPC_MSG variables managed by each element of the array is passed to TipcMsgDestroy(). When a TipcMsg array returned by a TipcMsg extraction routine is destroyed, the internal T_IPC_MSG variables managed by the array are passed to TipcMsgDestroy(). The reason for this is that a message or message array extracted from a message is read-only.

When a message is destroyed, any messages and message arrays within the original message are also destroyed.

## TipcMsg and T_IPC_MSG Reference Counts

The TipcMsg class makes use of the T_IPC_MSG reference count mechanism. The reference count of a message increments when creating and assigning TipcMsg objects. This saves memory, because two or more TipcMsg objects can reference the same T_IPC_MSG variable.

When a T_IPC_MSG variable is converted to a TipcMsg object, the appropriate constructor is used and the reference count of the message is incremented. For example:

```
T_IPC_MSG c_msg;

c_msg = TipcSrvNext(10.0);
if (c_msg != NULL) {
 // error
}
// At this point the message pointed to by c_msg has a reference
// count of 1.

TipcMsg msg(c_msg);
// Now the message has a reference count of 2.
```

However, when a TipcMsg object is converted to a T_IPC_MSG variable the reference count remains the same. For example:

```
T_IPC_MSG c_msg;

TipcMsg msg(T_MT_NUMERIC_DATA);
// At this point the reference count of the message is 1.

c_msg = msg;
// At this point the reference count of the message is still 1.
```

Whenever a TipcMsg object is assigned a new message, the original message managed by the TipcMsg object is passed to the C API TipcMsgDestroy() function. The effect of this is to decrement the reference count of the original message by one before assigning the new message to the TipcMsg object. There are no corresponding reference counts for TipcMt objects or objects that manage connection-oriented information (such as TipcConnClient or TipcConnServer).

If the reference count of the original message is one (1) at the time of the assignment, then TipcMsgDestroy() physically destroys the original message. See the detailed description of TipcMsgDestroy in the *TIBCO SmartSockets Application Programming Interface* reference for more information.

# Message Files

A message file is a representation of one or more messages. The content of the file is text or binary. It provides a means to represent messages in a file. See the *TIBCO SmartSockets User's Guide* for more information on the features of message files.

## Using Message Files

In the SmartSockets C++ class library, message files are managed by the TipcMsgFile class. There are two constructors from the TipcMsgFile class. The first constructor can be used to open an existing file for reading, to open an existing file for appending, or to create a new file for writing. For example:

```
TipcMsgFile msg_file("output.msg", T_IPC_MSG_FILE_CREATE_WRITE);
if (!msg_file) {
    // error
}
```

Messages are written to the file using the insertion operator (operator<<). For example:

```
TipcMsg msg(T_MT_NUMERIC_DATA);
msg << "Temperature" << (T_REAL8)98.6 << Check;
if (!msg) {
    // error
}

msg_file << msg;
if (!msg_file) {
    // error
}
```

The second constructor creates a TipcMsgFile object from an existing C FILE pointer.

The Check manipulator cannot be used with TipcMsgFile objects. In fact, the compiler generates an error if it is attempted. The Check manipulator can only be used in TipcMsg function chains.

Messages are read from a message file using the extraction operator (operator>>()). For example:

```
TipcMsgFile msg_file("input.msg", T_IPC_MSG_FILE_CREATE_READ);
if (!msg_file) {
    // error
}

TipcMsg msg;
msg_file >> msg;
if (!msg_file) {
    // error
}
```

# Chapter 4 Connections

All messages are transmitted between processes through connections. A connection is an endpoint of a direct communication link used to send and receive messages between two processes. The two processes, called peer processes, share the link. However, each process has a unique endpoint that it can manipulate independently. Connections can operate on messages in many different ways. Additional features of connections include:

- ability to use one of several different interprocess communications (IPC) protocols to transfer messages: TCP / IP or local (non-network)
- execution of callback functions at various points while operating on messages to perform user-defined actions
- detection of many kinds of network failures and take steps to recover from these failures
- request-reply communication capability using messages
- optional guaranteed message delivery (GMD)

  GMD stores copies of messages in files to enable total recovery from network failures.
- thread safety to use in multi-threaded programs

This chapter describes how to use connections and the relationship between connections and messages in the SmartSockets C++ class library. This chapter should be considered a companion to the *TIBCO SmartSockets User's Guide*. You can read detailed information on SmartSockets connections in the *TIBCO SmartSockets User's Guide*.

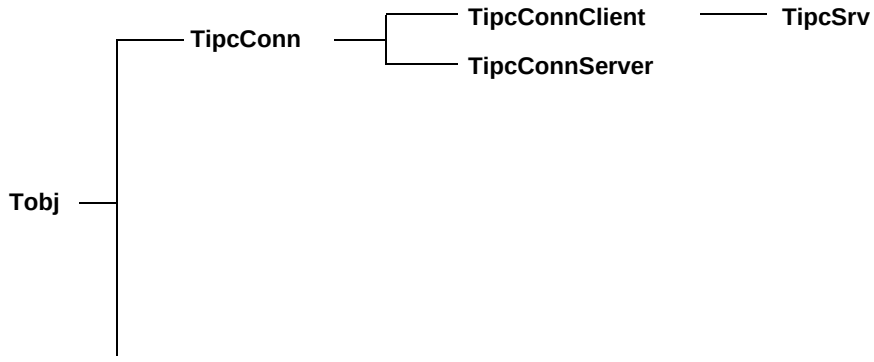
## Topics


- *Class Hierarchy Of Connections, page 42*
- *Working with Connections, page 43*

# Class Hierarchy Of Connections

The C++ class library organizes the various flavors of SmartSockets connections into several C++ classes (see Figure 4). The base class TipcConn declares and defines virtual member functions used by the derived user classes TipcConnServer and TipcConnClient. A final class derived from the TipcConnClient, called TipcSrv, which encapsulates the RTclient functionality, is described in Chapter 5, Publish and Subscribe.

*Figure 4 TipcConn Class Inheritance Hierarchy*




The TipcConn class contains a C type T_IPC_CONN and a collection of member functions that wrap the TipcConn* functions from the C API. Many of the member functions are declared as virtual member functions so that derived classes have the ability to reimplement their behavior. In fact, the TipcSrv class reimplements many of the TipcConn virtual member functions using TipcSrv* functions from the C API.

## Constructors

The proper constructor for a connection is selected depending on whether the process is acting as a server or a client.

### Server Connections

Server connections listen for connections from clients. They are created using the TipcConnServer class. Any of the supported protocols may be used with this class, which calls the C API function TipcConnCreateServer.

### Client Connections

The TipcConnClient class is used to create a connection that is hooked to the paired server connection. This class calls the C API function TipcConnCreateClient to make the connection and connect it to the corresponding server connection.

### Dummy Connections

The TipcConn class is used in instances where a dummy connection is desired. The member functions of the TipcConn class are capable of processing internally generated messages. This class creates a dummy connection by calling the C API function TipcConnCreate.

## Working with Connections

This section discusses how to create, access, and destroy a connection. To learn more about working with messages, see Chapter 3, Messages. This example programs show the C++ code required to send messages between two processes through a connection. The programs also show how to use each of the connection callback types using a static member function as a callback. For an example using conventional C functions as callbacks, see the code example in Working With The TipcSrv Class on page 63. There are two parts in this example: a server process and a client process.

The source code file for this example is located in these directories:

**UNIX:**

```
$RTHOME/examples/cxxipc
```

**OpenVMS:**

```
RTHOME:[EXAMPLES.CXXIPC]
```

**Windows:**

```
%RTHOME%\examples\cxxipc
```

## Server Source Code

The server waits for an client to connect to it, creates some callbacks, and then loops receiving and processing messages.

The server code is:

```
// conn_srv.cxx -- connections example RTserver

#include <iostream.h>
#include <rtworks/cxxipc.hxx>

// ---------------------------------------------------------------
// User handler class for various callbacks
// ---------------------------------------------------------------
class MyCallbacks {
public:
  MyCallbacks() {}
  virtual ~MyCallbacks() {}

  static void cb_process_numeric_data(T_IPC_CONN conn,
                                      T_IPC_CONN_PROCESS_CB_DATA data,
                                      T_CB_ARG arg);

  static void cb_default(T_IPC_CONN conn,
                         T_IPC_CONN_DEFAULT_CB_DATA data,
                         T_CB_ARG arg);

  static void cb_read(T_IPC_CONN conn,
                      T_IPC_CONN_READ_CB_DATA data,
                      T_CB_ARG arg);

  static void cb_queue(T_IPC_CONN conn,
                       T_IPC_CONN_QUEUE_CB_DATA data,
                       T_CB_ARG arg);

  static void cb_error(T_IPC_CONN conn,
                       T_IPC_CONN_ERROR_CB_DATA data,
                       T_CB_ARG arg);

};

// ===============================================================
//..MyCallbacks::cb_process_numeric_data -- numeric data callback
void
MyCallbacks::cb_process_numeric_data(T_IPC_CONN conn,
                                     T_IPC_CONN_PROCESS_CB_DATA data,
                                     T_CB_ARG arg)
{
  TutOut("Entering cb_process_numeric_data.\n");

  TipcMsg msg(data->msg);
```

```
  // set current field to first field in message
  msg.Current(0);
  if (!msg.Status()) {
    TutOut("Could not set current field of message.\n");
    return;
  }

  T_STR name;
  T_REAL8 value;

 // process all the fields of the message
  while (1) {
    msg >> name >> value;
    if (!msg) {
      break;
    }
    TutOut("%s = %f\n", name, value);
  }

  // make sure we reached the end of the message
  if (TutErrNumGet() != T_ERR_MSG_EOM) {
    TutOut("Did not reach end of message.\n");
  }

} // cb_process_numeric_data


// ================================================================
//..MyCallbacks::cb_default -- default callback
void MyCallbacks::cb_default(T_IPC_CONN conn,
                             T_IPC_CONN_DEFAULT_CB_DATA data,
                             T_CB_ARG arg)

{
  TutOut("Entering cb_default.\n");

  TipcMsg msg(data->msg);

  TipcMt mt(msg.Type());
  if (!mt) {
    TutOut("Could not get message type from message.\n");
    return;
  }


  T_STR name = mt.Name();
  if (!mt) {
    TutOut("Could not get name from message type.\n");
    return;
  }

  TutOut("Message type name is %s\n", name);

} // cb_default
```

```
// ================================================================
//..MyCallbacks::cb_read -- read callback
void MyCallbacks::cb_read(T_IPC_CONN conn,
                          T_IPC_CONN_READ_CB_DATA data,
                          T_CB_ARG arg)

{
  TutOut("Entering cb_read.\n");
  TipcMsg msg(data->msg);

  TipcMsgFile *output = (TipcMsgFile *)arg;
  *output << msg;

} // cb_read

// ================================================================
//..MyCallbacks::cb_queue -- queue callback
void MyCallbacks::cb_queue(T_IPC_CONN conn,
                           T_IPC_CONN_QUEUE_CB_DATA data,
                           T_CB_ARG arg)
{
  TutOut("Entering cb_queue.\n");

  TipcMsg msg(data->msg);

  if (!msg) {
    TutOut("MyCallbacks::cb_queue() Error constructing TipcMsg\n");
    return;
  }

  TipcMt mt(msg.Type());
  if (!mt) {
    TutOut("MyCallbacks::cb_queue() Error getting message type\n");
    return;
  }

  T_STR name = mt.Name();

  TutOut("A message of type %s is being handled\n", name);

 // print out the position of the message being inserted/deleted
  TutOut("A message of type %s is being %s at position %d.\n",
         name, data->insert_flag ? "inserted" : "deleted",
         data->position);

} // cb_queue
```

```
// ================================================================
//..MyCallbacks::cb_error -- error callback
void MyCallbacks::cb_error(T_IPC_CONN conn,
                           T_IPC_CONN_ERROR_CB_DATA data,
                           T_CB_ARG arg)

{
  TutOut("Entering cb_error.\n");
  TutOut("The error number is %d\n", data->err_num);
} // cb_error


// ================================================================
//..main -- main program
int main()
{
  TipcMt mt_vacant;

  TutOut("Creating server connection to accept clients on.\n");

  TipcConnServer accepting_server("tcp:_node:5252");
  if (!accepting_server) {
    TutOut("Could not create server connection for "
           "accepting clients.\n");
    return T_EXIT_FAILURE;
  }

  TutOut("Waiting for client to connect.\n");

  TipcConn *conn = accepting_server.Accept();

  if (!conn) {
    TutOut("Could not accept client.\n");
    return T_EXIT_FAILURE;
  }

  // create callbacks to be executed when certain operations occur
  TutOut("Create callbacks.\n");

  // create a process callback
  TipcMt mt(T_MT_NUMERIC_DATA);
  if (!mt) {
    TutOut("Could not look up NUMERIC_DATA message type.\n");
    return T_EXIT_FAILURE;
  }

  conn->ProcessCbLookup(mt,
                        MyCallbacks::cb_process_numeric_data,
                        NULL);
```

```
// check status flag of the object pointed to by conn
if (!*conn) {
  conn->ProcessCbCreate(mt,
                        MyCallbacks::cb_process_numeric_data,
                        NULL);
  if (!*conn) {
    TutOut("Could not create NUMERIC_DATA process callback.\n");
    return T_EXIT_FAILURE;
  }
}

// create a default callback
conn->DefaultCbLookup(MyCallbacks::cb_default, NULL);

if (!*conn) {
  conn->DefaultCbCreate(MyCallbacks::cb_default, NULL);

  if (!*conn) {
    TutOut("Could not create default callback.\n");
    return T_EXIT_FAILURE;
  }
}

// create a message file to use in read callback
TipcMsgFile *msg_file
  = new TipcMsgFile(stdout, T_IPC_MSG_FILE_CREATE_WRITE);

// create a read callback
conn->ReadCbLookup(mt_vacant, MyCallbacks::cb_read, msg_file);

if (!*conn) {
  conn->ReadCbCreate(mt_vacant, MyCallbacks::cb_read, msg_file);

  if (!*conn) {
    TutOut("Could not create read callback.\n");
    return T_EXIT_FAILURE;
  }
}


// create a queue callback
conn->QueueCbLookup(mt_vacant, MyCallbacks::cb_queue, NULL);

if (!*conn) {
  conn->QueueCbCreate(mt_vacant, MyCallbacks::cb_queue, NULL);

  if (!*conn) {
    TutOut("Could not create queue callback.\n");
    return T_EXIT_FAILURE;
  }
}
```

```
  // create an error callback
  conn->ErrorCbLookup(MyCallbacks::cb_error, NULL);

  if (!*conn) {
    conn->ErrorCbCreate(MyCallbacks::cb_error, NULL);

    if (!*conn) {
      TutOut("Could not create error callback.\n");
      return T_EXIT_FAILURE;
    }
  }

  // begin processing messages
  TutOut("Read and process all messages.\n");

  TipcMsg msg;  // message received and processed

  for (;;) {
    msg = conn->Next(T_TIMEOUT_FOREVER);
    if (!conn->Status()) {
      break;
    }

    conn->Process(msg);
    if (!conn) {
      TutOut("Could not process message.\n");
    }

    msg.Destroy();
  }

   // make sure we reached the end of the data
  if (TutErrNumGet() != T_ERR_EOF) {
    TutOut("Did not reach end of data.\n");
  }

  delete msg_file;
  delete conn;

  TutOut("Server process exiting successfully.\n");

  return T_EXIT_SUCCESS;  // all done

}  // main
```

## Client Source Code

The client process connects to the server process and sends two messages to the server. The client code is:

```
// conn_clt.cxx -- connections example client

#include <iostream.h>
#include <rtworks/cxxipc.hxx>

// A simple user C++ class encapsulating a write callback
class MyWriteCallback {
public:

  MyWriteCallback() {}
  virtual ~MyWriteCallback() {}

  static void cb_write(T_IPC_CONN conn,
                       T_IPC_CONN_WRITE_CB_DATA data,
                       T_CB_ARG arg);
};

// ====================================================================
//..MyWriteCallback::cb_write -- write callback
void MyWriteCallback::cb_write(T_IPC_CONN conn,
                               T_IPC_CONN_WRITE_CB_DATA data,
                               T_CB_ARG arg)
{
  TutOut("Entering cb_write.\n");
  TipcMsg msg(data->msg);

   // print out the message to the message file

  TipcMsgFile *output = (TipcMsgFile *)arg;
  *output << msg;
} // cb_write

// ====================================================================
//..main -- main program
int main()
{
  TutOut("Creating connection to server process.\n");

  TipcConnClient client("tcp:_node:5252");

  if (!client) {
    TutOut("Could not create connection to server.\n");
    return T_EXIT_FAILURE;
  }

  // create callbacks to be executed when certain operations occur
  TutOut("Create callbacks.\n");

  TipcMsgFile *msg_file
    = new TipcMsgFile(stdout, T_IPC_MSG_FILE_CREATE_WRITE);
```

```
// create write callback
TipcMt mt_vacant;
client.WriteCbLookup(mt_vacant, MyWriteCallback::cb_write,
                     msg_file);

if (!client) {
  client.WriteCbCreate(mt_vacant, MyWriteCallback::cb_write,
                       msg_file);

  if (!client) {
    TutOut("Could not create write callback.\n");
    return T_EXIT_FAILURE;
  }
}


TutOut("Constructing and sending a NUMERIC_DATA message.\n");

TipcMsg msg(T_MT_NUMERIC_DATA);

if (!msg) {
  TutOut("Could not create NUMERIC_DATA message.\n");
  return T_EXIT_FAILURE;
}

msg << "voltage" << (T_REAL8)33.4534
    << "switch_pos" << (T_REAL8)0.0 << Check;

if (!msg) {
  TutOut("Could not append fields to NUMERIC_DATA message\n");
  return T_EXIT_FAILURE;
}
if (!client.Send(msg)) {
  TutOut("Could not send NUMERIC_DATA message.\n");
}

TutOut("Constructing and sending an INFO message.\n");

TipcMt mt(T_MT_INFO);
if (!mt) {
  TutOut("Could not look up INFO message type.\n");
  return T_EXIT_FAILURE;
}

if (!msg.Type(mt)) {
  TutOut("Could not set message type.\n");
  return T_EXIT_FAILURE;
}

if (!msg.NumFields(0)) {
  TutOut("Could not set message num fields.\n");
  return T_EXIT_FAILURE;
}
```

```
  msg << "Now is the time";

  if (!msg) {
    TutOut("Could not append fields to INFO message.\n");
    return T_EXIT_FAILURE;
  }

  if (!client.Send(msg)) {
    TutOut("Could not send INFO message.\n");
    return T_EXIT_FAILURE;
  }

  if (!client.Flush()) {
    TutOut("Could not flush buffered outgoing messages\n");
    return T_EXIT_FAILURE;
  }

  delete msg_file;

  TutOut("Client process exiting successfully.\n");
  return T_EXIT_SUCCESS;   // all done

}  // main
```

## Running Server and Client Programs

Compile and link the server and client programs (see Compiling, Linking, and Running on page 13 for generic instructions).

To run the programs, start the server process first in one terminal emulator window and then the client process in another terminal emulator window.

Start up the RTserver program in the first window:

**UNIX:**

```
$ conn_srv.x
```

**OpenVMS:**

```
$ run conn_srv.exe
```

**Windows:**

```
$ conn_srv.exe
```

Start up the client program in the second window:

**UNIX:**

```
$ conn_clt.x
```

**OpenVMS:**

```
$ run conn_clt.exe
```

**Windows:**

```
$ conn_clt.exe
```

The output from the server program is similar to this:

```
Creating server connection to accept clients on.
Waiting for client to connect.
Create callbacks.
Read and process all messages.
Entering cb_read.
numeric_data _null {
   voltage 33.4534
   switch_pos 0
}
Entering cb_queue.
A message of type numeric_data is being handled
A message of type numeric_data is being inserted at position 0.
Entering cb_read.
info _null "Now is the time"
Entering cb_queue.
A message of type info is being handled
A message of type info is being inserted at position 1.
Entering cb_queue.
A message of type numeric_data is being handled
A message of type numeric_data is being deleted at position 0.
Entering cb_process_numeric_data.
   voltage = 33.453400
   switch_pos = 0.000000
Entering cb_queue.
A message of type info is being handled
A message of type info is being deleted at position 0.
Entering cb_default.
Message type name is info
Entering cb_error.
The error number is 10
Server process exiting successfully.
```

The output from the client program follows.

```
Creating connection to server process.
Create callbacks.
Constructing and sending a NUMERIC_DATA message.
Entering cb_write.
numeric_data _null {
   voltage 33.4534
   switch_pos 0
}
Constructing and sending an INFO message.
Entering cb_write.
info _null "Now is the time"
Client process exiting successfully.
```

## Creating Connections

As stated in the *TIBCO SmartSockets User's Guide*, there are three steps required for two processes to connect to each other in a client-server model:

1. Creating a server connection.
2. Creating a client connection.
3. Accepting the client on the server's part.

### Creating a Server Connection

The server connection, using any one of the supported protocols, is created by using the TipcConnServer constructor.

The TipcConnServer class has two constructors. The first constructor takes a logical connection name as its only parameter. For more information see the *TIBCO SmartSockets User's Guide*.

| Protocol | Constructor Parameter |
|---|---|
| Local | `local:`*node*`:`*name*<br>where *name* is a filename |
| TCP / IP | `tcp:`*node*`:`*port_number*<br>a service name may also be used in place of *port_number* |

The only purpose of a server connection is to accept client connections (using the Accept() member function). Messages cannot be sent or received on a server connection.

The Accept() member function returns a pointer to a new TipcConn object, which refers to one end point of a peer-to-peer connection. It only sends and receives messages.

The preferred constructor is the preceding constructor. However, a second constructor is provided in case it is needed in an application under special circumstances. The second constructor takes a C type T_IPC_CONN, which is created with TipcConnCreateServer, as an argument. For example:

```
server_conn = TipcConnCreateServer("tcp:_node:5252");
if (server_conn == NULL) {
   TutOut("Could not create server connection.\n");
   return T_EXIT_FAILURE;
}
TipcConnServer server_conn_obj(server_conn, FALSE);
```

This constructor may also take an optional second parameter as shown in the preceding example. The second parameter sets a flag inside the object as to whether the connection passed in the first parameter should be destroyed when the object's destructor is invoked. A value of TRUE is used to destroy the connection when the destructor is invoked, and a value of FALSE is used to leave the connection as-is when the destructor is invoked.

### Creating a Client Connection

In a fashion similar to server connections, a client connection is created by using the TipcConnClient constructor:

```
TipcConnClient conn("tcp:_node:5252");
if (!conn) {
   TutOut("Could not create connection to the server.\n");
   return T_EXIT_FAILURE;
}
```

The client connection must use the same IPC protocol as the server. The client connection must be created after the server connection, as the client needs something to contact.

The preferred constructor is the preceding constructor. However, a second constructor is provided in case it is needed in a project under special circumstances. The second constructor takes a C type T_IPC_CONN, which is created with TipcConnCreateClient, as an argument. For example:

```
conn = TipcConnCreateClient("tcp:_node:5252");
if (conn == NULL) {
   TutOut("Could not create connection to the server.\n");
   return T_EXIT_FAILURE;
}
TipcConnClient conn_obj(conn, FALSE);
```

This constructor may also take an optional second parameter as shown in the example above. The second parameter sets a flag inside the object as to whether the connection passed in the first parameter should be destroyed when the object's destructor is invoked. A value of TRUE is used to destroy the connection when the destructor is invoked, and a value of FALSE is used to leave the connection as-is when the destructor is invoked.

### Accepting the Client

Once the client has constructed a TipcConnClient object to create its connection, the server process must accept the client by calling the member function Accept(). For example:

```
TipcConn *client_conn = server_conn.Accept();
if (client_conn == NULL) {
   TutOut("Could not accept client.\n");
   return T_EXIT_FAILURE;
}
```

Note that the Accept() member function returns a pointer to a new TipcConn object.

## Destroying a Connection

In most cases, a connection is destroyed when the managing object's destructor is called. The only exception to this is when a managing object is created using an existing C type of T_IPC_CONN as the constructor's first argument and FALSE as the second argument. (This constructor is referred to as the second constructor in the discussions above.)

The C API TipcConnDestroy() function should not be used with any class derived from TipcConn, because TipcConnDestroy() renders TipcConn objects unusable. This happens because TipcConnDestroy() destroys the memory pointed to by the internal T_IPC_CONN pointer of TipcConn. Once that memory is gone, the TipcConn object has no data on which to operate. For example, this code fragment leaves a TipcConnClient object in an unusable state:

```
TipcConnClient conn("tcp:_node:5252");
if (!conn) {
   TutOut("Could not connect to server\n");
   return T_EXIT_FAILURE;
}

if (!TipcConnDestroy(conn)) {
   TutOut("Could not destroy connection.\n");
}
// At this point, conn is unusable
```

## Callbacks

Callbacks are functions that are executed when certain operations occur. Callbacks are conceptually equivalent to dynamically adding a line of code to a program. For more discussion on callback functions, see the *TIBCO SmartSockets Utilities* reference.

When the C API to connections is used, a callback function is created using the function whose name takes the form TipcConnTypeCbCreate. A callback function is looked up using the function TipcConnTypeCbLookup and then is manipulated with one of the TutCb* functions.

When the C++ class library is used, callbacks are created by calling a member function of a derived class of TipcConn whose name has the form TypeCbCreate. The code example in Working with Connections on page 43 begins by grouping all the callback functions (cb_process_numeric_data, cb_default, cb_read, cb_queue, and cb_error) together into a single C++ class called MyCallbacks. Callback functions are looked up by using a member function of a derived class of TipcConnWrapper whose name has the form TypeCbLookup.

In general, the derived TipcConn classes do not require callbacks to be C++ class member functions, so you are free to use conventional C functions as callbacks. However, if a C++ member function is to be used as a callback, it must be declared static, as in the example found in Working with Connections on page 43. For an example using conventional C functions as callbacks, see the code example in Working With The TipcSrv Class on page 63.

### Global Callbacks

In the C API, a global callback is created by passing a `NULL` in the T_IPC_MT argument of a create callback function. For example, to create a global connection read callback using the C API, do this:

```
if (!TipcConnReadCbCreate(conn, NULL, my_conn_read_cb, some_data))
{
    // error
}
```

In C++, a vacant TipcMt object creates a global callback. For example:

```
TipcMt mt_vacant;

conn.ReadCbCreate(mt_vacant, my_conn_read_cb, some_data);
if (!conn) {
    // error
}
```

# Chapter 5 Publish and Subscribe

This chapter introduces RTserver and RTclient, which allow many processes to easily communicate with each other using a publish-subscribe communication model. An RTserver process routes messages between RTclient processes. A key feature of SmartSockets is the ability to distribute RTserver processes and RTclient processes over a network. Different processes can be run on different computers, taking advantage of all the computing power a network has to offer. RTservers and RTclients can be dynamically started and stopped while the system is running.

## Topics


- *Overview, page 60*
- *RTserver and RTclient Composition, page 61*
- *Working With The TipcSrv Class, page 63*

## Overview

The functionality of RTserver and RTclient is layered on top of connections and messages, but adds greater functionality and ease of use. Some of these functions are:

- RTserver and RTclient have simplified setup and control through options, which require no programming
- RTserver can partition a group of RTclients into a project
- RTserver and RTclient use logical addresses called subjects for the sender property and destination property of messages, which enable powerful yet simple publish-subscribe services
- Subjects are arranged in a hierarchical namespace with wildcard capabilities, which makes it easier to build large projects
- A group of RTservers can distribute the load of publish-subscribe message routing with dynamic message routing that offers greater scalability and flexibility
- An RTclient can automatically start an RTserver, automatically restart an RTserver, and even continue running when an RTserver is temporarily unavailable
- RTserver and RTclient have advanced guaranteed message delivery (GMD) capabilities, such as automatic recovery from most network failures
- Much information about RTservers and RTclients can be monitored, which allows you to easily debug, examine, and control your projects
- Load balancing can be used to publish messages to one subscriber instead of all
- The RTclient API can be used safely in multi-threaded programs

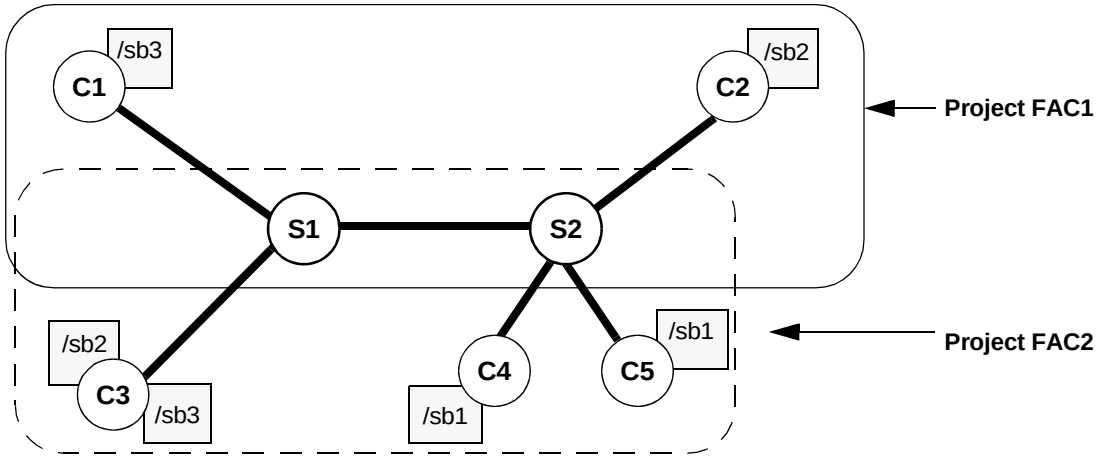
This chapter describes the C++ interface to RTserver and how to work with the RTclient C++ API.

This chapter should be considered a companion to the *TIBCO SmartSockets User's Guide*. You may wish to read the detailed information on RTserver and RTclient processes in the *TIBCO SmartSockets User's Guide* before proceeding further in this chapter. The intent of this chapter is to focus on creating RTclient processes using the C++ class library.

# RTserver and RTclient Composition

Before you use RTserver and the RTclient C++ interfaces, it helps to have an understanding of the concepts involved. Figure 5 shows the RTserver and RTclient architecture.

*Figure 5 RTserver and RTclient Architecture*




Note

Project FAC1 has processes C1, S1, S2, and C2.

Project FAC2 has processes C3, S1, C4, S2, and C5.

Both RTserver S1 and S2 are used by both projects.

RTclient C1 cannot send messages to C3, C4, or C5 because they are not in the same project.

A message published (sent) to the /sb1 subject in project FAC2 is received by both RTclient C4 and C5.

= a connection

S = RTserver process

C = RTclient process

/sb = a subject being subscribed to by RTclient

This architecture is made up of these major concepts:

| | |
|---|---|
| RTserver | A process that extends the features of connections to provide transparent publish-subscribe message routing among many processes. |
| RTclient | Any program (user-defined or SmartSockets client) that connects to RTserver and accesses its services (under this definition RTmon can be considered an RTclient). |
| Project | A group of RTservers and RTclients working together. |

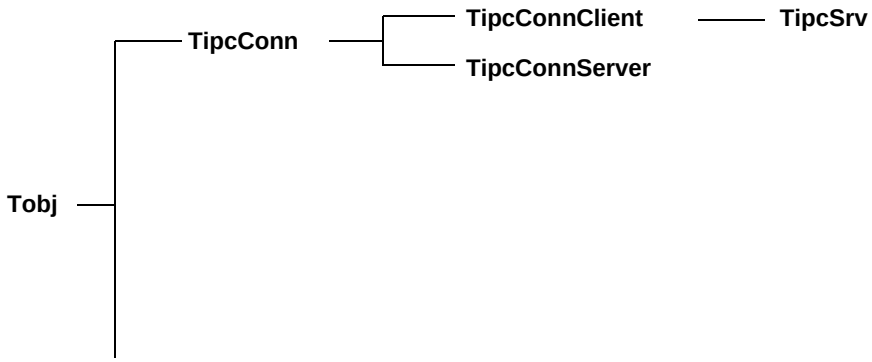
| | |
|---|---|
| Subject | A logical address for a message; RTclient subscribes to (registers interest in) subjects; an RTclient also publishes (sends) messages to subjects. |
| Data Frame | A group of messages with the same timestamp (not shown in the figure). |
| Monitoring | Allows you to examine detailed information about your project in real time (not shown in the figure). |

More detailed information can be found in the *TIBCO SmartSockets User's Guide*.

## The TipcSrv Class

The SmartSockets C++ class TipcSrv creates RTclient processes that connect to RTserver. TipcSrv is derived from the base class TipcConnClient.

*Figure 6 TipcConn Class Inheritance Hierarchy*




Because most SmartSockets C API TipcConn* calls have a corresponding TipcSrv* equivalent that only operates on the connection to RTserver, many of the member functions in TipcConn are declared virtual, and the TipcSrv class reimplements them appropriately in its derived class. The C API TipcConn* functions that do not have TipcSrv* equivalents, such as TipcConnAccept(), do not appear in TipcConn, but rather appear in derived classes of TipcConn other than TipcSrv (such as TipcConnServer).

Because any given RTclient process is allowed to have, at most, one connection to RTserver at a time, the SmartSockets C++ class library does not allow you to construct an instance of a TipcSrv object. However, sometimes you may need to cause the C API's TipcSrvCreate() to be called from C++ using some type of mechanism. Use the static member functions on the TipcSrv class, such as TipcSrv::InstanceCreate(), to obtain the TipcSrv functionality and optionally establish a connection to an RTserver process.

You can establish a full connection to RTserver in a single step or in two steps. This example illustrates connecting to RTserver in one step:

```
TipcSrv& server = TipcSrv::InstanceCreate(T_IPC_SRV_CONN_FULL);
```

The next example illustrates connecting to RTserver in two steps.

```
TipcSrv& server = TipcSrv::Instance();
if (server.CreateCbLookup(create_cb, NULL) == NULL) {
   server.CreateCbCreate(create_cb, NULL);
}
status = server.Create(T_IPC_SRV_CONN_FULL);
if (!status) {
   // error
}
```

The advantage of calling TipcSrv::InstanceCreate() over the combination of TipcSrv::Instance() and TipcSrv::Create() is that TipcSrv::InstanceCreate() returns a handle to a TipcSrv object and creates a connection to RTserver in one step. The advantage of using the combination of TipcSrv::Instance() and TipcSrv::Create() is that TipcSrv::Instance() returns a reference to a TipcSrv object, which in turn is used to call TipcSrv member functions, such as TipcSrv::CreateCbCreate(), prior to connecting to RTserver with the TipcSrv::Create().

## Working With The TipcSrv Class

These two programs show code used to send messages between two RTclient processes through RTserver using subjects. They also illustrate using regular C functions as callback functions. For an example using static member functions of a user defined class as callback functions, see the code example in Working with Connections on page 43.

The source code file for this example is located in this directory:

**UNIX:**

```
$RTHOME/examples/cxxipc
```

**OpenVMS:**

```
RTHOME:[EXAMPLES.CXXIPC]
```

**Windows:**

```
%RTHOME%\examples\cxxipc
```

## Common Header Source Code

The code for a common header file is:

```
// rtclient.hxx -- common header for RTclient examples

#define EXAMPLE_PROJECT "example"
#define EXAMPLE_SUBJECT "rcv"
```

## Receiver Side

The receiving RTclient creates its connection to RTserver, and receives and processes messages. The receiver code is:

```
// rtcltrcv.cxx -- RTclient example receiver

// The receiving RTclient creates its connection to RTserver, and
// receives and processes messages.
#include <rtworks/cxxipc.hxx>
#include "rtclient.hxx"

// ================================================================
//..cb_process_numeric_data -- numeric data callback
void cb_process_numeric_data(T_IPC_CONN conn,
                             T_IPC_CONN_PROCESS_CB_DATA data,
                             T_CB_ARG arg)
{
  TutOut("Entering cb_process_numeric_data.\n");

  TipcMsg msg(data->msg);

  // set current field to first field in message
  msg.Current(0);
  if (!msg.Status()) {
    TutOut("Could not set current field of message.\n");
    return;
  }

  T_STR name;
  T_REAL8 value;

  while (1) {
    msg >> name >> value;
    if (!msg.Status()) {
      break;
    }
    TutOut("%s = %f\n", name, value);
  }

  // make sure we reached the end of the message
  if (TutErrNumGet() != T_ERR_MSG_EOM) {
    TutOut("Did not reach end of message.\n");
  }
} // cb_process_numeric_data
```

```
// ====================================================================
//..cb_default -- default callback
void cb_default(T_IPC_CONN conn,
                T_IPC_CONN_DEFAULT_CB_DATA data,
                T_CB_ARG arg)
{
  TutOut("Entering cb_default.\n");

  TipcMsg msg(data->msg);
  TipcMt mt(msg.Type());
  if (!mt) {
    TutOut("Could not get message type from message.\n");
    return;
  }

  T_STR name =  mt.Name();
  if (!mt) {
    TutOut("Could not get name from message type.\n");
    return;
  }

  TutOut("Message type name is %s\n", name);

} // cb_default

// ====================================================================
//..main -- main program
int main(int argc, char **argv)
{
  T_OPTION option;
  TipcMt mt;

  // Set the option Project to partition ourselves.
  option = TutOptionLookup("project");
  if (option == NULL) {
    TutOut("Could not look up the option named project.\n");
    return T_EXIT_FAILURE;
  }
  if (!TutOptionSetEnum(option, EXAMPLE_PROJECT)) {
    TutOut("Could not set the option named project.\n");
    return T_EXIT_FAILURE;
  }

 // Allow a command-line argument containing the name of a
 // SmartSockets startup command file. This file can be used to set
 // options like Server_Names.
  if (argc == 2) {
    if (!TutCommandParseFile(argv[1])) {
      TutOut("Could not parse startup command file %s.\n",
argv[1]);
      return T_EXIT_FAILURE;
    }
  }
```

```
  else if (argc != 1) { // too many command-line arguments
    TutOut("Usage: %s [ command_file_name ]\n", argv[0]);
    return T_EXIT_FAILURE;
  }

  TutOut("Creating connection to RTserver.\n");
  TipcSrv& rcvr = TipcSrv::InstanceCreate(T_IPC_SRV_CONN_FULL);
  if (!rcvr) {
    TutOut("Could not create the connection to RTserver.\n");
    return T_EXIT_FAILURE;
  }

 // create callbacks to be executed when certain operations occur
  TutOut("Create callbacks.\n");

 // process callback for NUMERIC_DATA
  mt.Lookup(T_MT_NUMERIC_DATA);
  if (!mt) {
    TutOut("Could not look up NUMERIC_DATA message type.\n");
    return T_EXIT_FAILURE;
  }
  if (rcvr.ProcessCbCreate(mt,
                           cb_process_numeric_data, NULL) == NULL)
{
    TutOut("Could not create NUMERIC_DATA process callback.\n");
    return T_EXIT_FAILURE;
  }

 // default callback
  if (rcvr.DefaultCbCreate(cb_default, NULL) == NULL) {
    TutOut("Could not create default callback.\n");
    return T_EXIT_FAILURE;
  }

  TutOut("Start subscribing to standard subjects.\n");
  if (!rcvr.StdSubjectSetSubscribe(TRUE, FALSE)) {
    TutOut("Could not start subscribing to standard subjects.\n");
    return T_EXIT_FAILURE;
  }

  TutOut("Start subscribing to the %s subject.\n",
EXAMPLE_SUBJECT);
  if (!rcvr.SubjectSubscribe(EXAMPLE_SUBJECT, TRUE)) {
    TutOut("Could not start subscribing to the %s subject.\n",
           EXAMPLE_SUBJECT);
    return T_EXIT_FAILURE;
  }

 // If an error occurs, then TipcSrv::MainLoop will restart RTserver
 // and return FALSE. We can safely continue.
  for (;;) {
    if (!rcvr.MainLoop(T_TIMEOUT_FOREVER)) {
      TutOut("TipcSrv::MainLoop failed with error <%s>.\n",
             TutErrStrGet());
    }
  }
```

```
 // This line should not be reached.
  TutOut("This line should not be reached!!!\n");
  return T_EXIT_FAILURE;

} // main
```

## Sender Side

This sending RTclient creates its connection and publishes messages to a subject (through RTserver). The sending code is:

```
// rtcltsnd.cxx -- RTclient example sender

// This sending RTclient creates its connection and sends a data frame
// of messages to a subject (through RTserver).

#include <rtworks/cxxipc.hxx>
#include "rtclient.hxx"

// ====================================================================
//..main -- main program
int main(int argc, char **argv)
{
  T_OPTION option;
  TipcMt mt;

  // Set the option Project to partition ourselves.
  option = TutOptionLookup("project");
  if (option == NULL) {
    TutOut("Could not look up the option named project.\n");
    return T_EXIT_FAILURE;
  }
  if (!TutOptionSetEnum(option, EXAMPLE_PROJECT)) {
    TutOut("Could not set the option named project.\n");
    return T_EXIT_FAILURE;
  }

 // Log outgoing data messages to a message file. Another
 // way to set options is to use TutCommandParseStr.
  TutCommandParseStr("setopt log_out_data log_out.msg");

 // Allow a command-line argument containing the name of a
 // SmartSockets startup command file. This file can be used to set
 // options like Server_Names.
  if (argc == 2) {
    if (!TutCommandParseFile(argv[1])) {
      TutOut("Could not parse startup command file %s.\n",
             argv[1]);
      return T_EXIT_FAILURE;
    }
  }
```

```
  else if (argc != 1) { // too many command-line arguments
    TutOut("Usage: %s [ command_file_name ]\n", argv[0]);
    return T_EXIT_FAILURE;
  }

  TutOut("Creating connection to RTserver.\n");
  TipcSrv& sndr = TipcSrv::InstanceCreate(T_IPC_SRV_CONN_FULL);
  if (!sndr) {
    TutOut("Could not create the connection to RTserver.\n");
    return T_EXIT_FAILURE;
  }

  TutOut("Send a frame of data to the subject.\n");

  TutOut("Sending a TIME message.\n");
  mt.Lookup(T_MT_TIME);
  if (!mt) {
    TutOut("Could not look up TIME message type.\n");
    return T_EXIT_FAILURE;
  }
  if (!sndr.SrvWrite(EXAMPLE_SUBJECT, mt, TRUE,
                     T_IPC_FT_REAL8, 1.0,
                     NULL)) {
    TutOut("Could not send TIME message.\n");
  }
TutOut("Sending a NUMERIC_DATA message.\n");
  mt.Lookup(T_MT_NUMERIC_DATA);
  if (!mt) {
    TutOut("Could not look up NUMERIC_DATA message type.\n");
    return T_EXIT_FAILURE;
  }
  if (!sndr.SrvWrite(EXAMPLE_SUBJECT, mt, TRUE,
                     T_IPC_FT_STR, "voltage",
                     T_IPC_FT_REAL8, 33.4534,
                     T_IPC_FT_STR, "switch_pos",
                     T_IPC_FT_REAL8, 0.0,
                     NULL)) {
    TutOut("Could not send NUMERIC_DATA message.\n");
  }

  TutOut("Sending an END_OF_FRAME message.\n");
  mt.Lookup(T_MT_END_OF_FRAME);
  if (!mt) {
    TutOut("Could not look up END_OF_FRAME message type.\n");
    return T_EXIT_FAILURE;
  }
  if (!sndr.SrvWrite(EXAMPLE_SUBJECT, mt, TRUE, NULL)) {
    TutOut("Could not send END_OF_FRAME message.\n");
  }
```

```
    // Each RTclient automatically creates a connection process
    // callback for CONTROL messages. Use this to send the command
    // "quit force" to the receiver's command interface.
  TutOut("Sending a CONTROL message to stop the receiver(s).\n");
  mt.Lookup(T_MT_CONTROL);
  if (!mt) {
    TutOut("Could not look up CONTROL message type.\n");
    return T_EXIT_FAILURE;
  }
  if (!sndr.SrvWrite(EXAMPLE_SUBJECT, mt, TRUE,
                     T_IPC_FT_STR, "quit force",
                     NULL)) {
    TutOut("Could not send CONTROL message.\n");
  }

 // Flush the buffered outgoing messages to RTserver.
  if (!sndr.Flush()) {
    TutOut("Could not flush outgoing messages to RTserver.\n");
  }

 // Completely disconnect from RTserver.

  if (!sndr.Destroy(T_IPC_SRV_CONN_NONE)) {
    TutOut("Could not fully destroy connection to RTserver.\n");
  }
  TutOut("Sender RTclient process exiting successfully.\n");
  return T_EXIT_SUCCESS; // all done

} // main
```

## Running Receiving and Sending Programs

Compile and link the receiving and sending programs (see Compiling, Linking, and Running on page 13 for generic instructions).

To run the programs, start the receiving process first in one terminal emulator window and then the sending process in another terminal emulator window.

Start RTserver, then start up the receiving program in the first window:

**UNIX:**

```
$ rtcltrcv.x
```

**OpenVMS:**

```
$ run rtcltrcv.exe
```

**Windows:**

```
$ rtcltrcv.exe
```

Start up the sending program in the second window:

**UNIX:**

```
$ rtcltsnd.x
```

**OpenVMS:**

```
$ run rtcltsnd.exe
```

**Windows:**

```
$ rtcltsnd.exe
```

The output from the receiving process is similar to this:

```
Creating connection to RTserver.
Connecting to project <example> on <_node> RTserver.
Using local protocol.
Message from RTserver: Connection established.
Start subscribing to subject </_workstation1_7704>.
Create callbacks.
Start subscribing to standard subjects.
Start subscribing to subject </_workstation1>.
Start subscribing to subject </_all>.
Start subscribing to the rcv subject.
Entering cb_default.
Message type name is time
Entering cb_process_numeric_data.
voltage = 33.453400
switch_pos = 0.000000
Entering cb_default.
Message type name is end_of_frame
```

The output from the sending process is similar to this:

```
Now logging outgoing data-related messages to <log_out.msg>.
Creating connection to RTserver.
Connecting to project <example> on <_node> RTserver.
Using local protocol.
Message from RTserver: Connection established.
Start subscribing to subject </_workstation1_7706>.
Send a frame of data to the receiving subject.
Sending a TIME message.
Sending a NUMERIC_DATA message.
Sending an END_OF_FRAME message.
Sending a CONTROL message to stop the receiver(s).
Sender RTclient process exiting successfully.
```

This is the message file `log_out.msg`, which is created by the sender:

```
time /rcv 1
numeric_data /rcv {
 voltage 33.4534
 switch_pos 0
}
end_of_frame /rcv
control /rcv "quit force"
```

# Chapter 6 C++ Class Reference

When working in C++, you interact with the SmartSockets C++ Class Library in several ways:

- By constructing a C++ object using an appropriate class constructor, then invoking the member functions of that object
- By invoking static member functions of a class without necessarily constructing any particular objects of that class
- By deriving new C++ classes as subtypes of provided classes in order to extend or otherwise modify the behavior of SmartSockets classes

This chapter describes the C++ class library in detail on a class-by-class basis. The *TIBCO SmartSockets User's Guide* and the *TIBCO SmartSockets Application Programming Interface*, which are companion reference documents to this manual, describe RTserver and all other features using the C programming language (Tipc) API. Because most SmartSockets class member functions are simple C++ wrappers around a corresponding C language Tipc API function, this manual does not include an exhaustive description of each member function's diagnostics, error behavior, return values, or points of caution on its use, except where they differ from the Tipc behavior. Instead, this document refers you to appropriate entries in the companion manual.

## Topics


- *Class Overview, page 73*
- *Data Structures, page 75*
- *Include Files, page 76*
- *Utility Functions and Macros, page 77*
- *Compiling and Linking, page 77*
- *Error Handling, page 79*
- *C++ Class Descriptions, page 80*
- *TipcConn, page 81*

- *TipcConnClient, page 95*
- *TipcConnServer, page 97*
- *TipcMon, page 99*
- *TipcMonClient, page 104*
- *TipcMonServer, page 108*
- *TipcMsg, page 110*
- *TipcMsgFile, page 158*
- *TipcMt, page 161*
- *TipcSrv, page 168*
- *Tobj, page 185*

# Class Overview

Each class provided in the C++ Class Library is either an abstract base class or a user class. Abstract base class instances are not directly constructible using the SmartSockets C++ Class Library. Instead, you construct instances of an appropriate derived user class. All the classes are summarized here:

| Class | Description |
|---|---|
| TipcConnServer | User class representing a SmartSockets server connection. Implements the behavior found in the C API function TipcConnCreateServer. |
| TipcConnClient | User class representing a SmartSockets client connection. Implements the behavior found in the C API function TipcConnCreateClient. |
| TipcMon | User class for interacting with general monitoring information. |
| TipcMonClient | User class for interacting with RTclient monitoring information. |
| TipcMonServer | User class for interacting with RTserver monitoring information. |
| TipcMsg | User class for constructing, manipulating, and destroying a SmartSockets message.<br>• Encapsulates information about a SmartSockets message, including accessors for its type, sender, destination, priority, delivery mode, read-only, and other properties.<br>• Defines accessors for manipulating the data property of a SmartSockets message. |
| TipcMsgFile | User class for file I/O operations such as TipcMsgFileWrite. |
| TipcMt | User class for constructing, manipulating, looking up, and destroying SmartSockets message types. |
| TipcSrv | User class representing the RTclient functionality (TipcSrv APIs). Manages no private TipcSrv-specific data, but does include a static instance pointer to permit one-time object creation only (SmartSockets processes do not allow more than one RTserver connection per process). |
| Tobj | Abstract base class for error-handling, containing a status flag only. |

## Relation to the C API

Both the C++ class library and the C API upon which the C++ class library is built allow you to send and receive messages asynchronously across a heterogeneous computer network. They can be used to perform tasks such as:

- sending and receiving prioritized messages between two processes through connections
- executing user-defined callback functions when certain operations occur
- sending messages among many processes by RTserver using subjects
- monitoring processes by watching the subjects to which they are subscribing
- retrieving the current values of all slots in a subject
- detecting network failures and initiating actions to recover from them
- sending messages using different IPC protocols over a heterogeneous network of computers

The C++ class library offers a C++ member function for most of the SmartSockets C API calls.

## Naming Conventions

The C language IPC API uses these naming conventions:

- All IPC functions start with Tipc
- All SmartSockets message type numbers start with T_MT_
- Most IPC typedefs, enumerated values, and defines start with T_IPC_
- All message functions start with TipcMsg
- All message type functions start with TipcMt
- All connection functions start with TipcConn
- All functions that communicate with RTserver start with TipcSrv
- All monitoring functions start with TipcMon

The C++ class library extends these conventions:

- C++ public member functions start with uppercase letters, and their names appear with their redundant C API prefixes stripped from them. For example, the C API function TipcMtLookup becomes TipcMt::Lookup in the C++ class library.
- C++ private and protected member functions start with lowercase letters. For example, TipcMsg::size() in the C++ class library.

- Where the C API functions appear in `set` and `get` pairs, the C++ class library replaces them with C++ polymorphic, same-named pairs, differentiated by whether they require an argument or not. For example, the C API functions called TipcMsgSetType and TipcMsgGetType become the member functions TipcMsg::Type(TipcMt& *mt*) and TipcMsg::Type() in the C++ class library.
- Some C++ member functions truncate the C API name when it is not ambiguous. For example, the C API function TipcSrvMsgSend becomes TipcSrv::Send in the C++ class library.

## Data Structures

The C language API makes a number of IPC data structures accessible from user-written programs. These structures are defined (typedefed) in `<rtworks/ipc.h>`:

```
typedef struct T_IPC_MT_STRUCT T_IPC_MT_STRUCT, *T_IPC_MT;
typedef struct T_IPC_MSG_STRUCT T_IPC_MSG_STRUCT, *T_IPC_MSG;
typedef struct T_IPC_MSG_FIELD_STRUCT T_IPC_MSG_FIELD_STRUCT,
                 *T_IPC_MSG_FIELD;
typedef struct T_IPC_MSG_FILE_STRUCT T_IPC_MSG_FILE_STRUCT,
                 *T_IPC_MSG_FILE;
typedef struct T_IPC_CONN_STRUCT T_IPC_CONN_STRUCT, *T_IPC_CONN;
```

These structure definitions are incomplete, in the sense that none of the fields in the structure are defined. This is one way of creating an opaque data type in C. Pointers to these structures cannot be dereferenced to look at the data where they are pointing. These pointers are only manipulated through access functions, which provide binary compatibility that protects you from changes to the underlying data structures from one version of SmartSockets to the next.

The C++ classes for the most part eliminate the need for direct manipulation of these IPC data structures. For example, C++ programs may replace the T_IPC_MT class with the TipcMt class, and the T_IPC_MSG class with the TipcMsg class. However, on some occasions, most notably at object construction time and in callback functions, the C++ class library requires data structure manipulation. The example code fragments in the previous chapters and in the manual pages later in this chapter are a good source for learning these exception cases and how to treat them in C++.

# Include Files

Several important header files are needed to compile and link with SmartSockets. These files are normally included in C/C++ code with the #include C/C++ preprocessor directive.The SmartSockets header files are located in these directories:

**UNIX:**

`$RTHOME/include/$RTARCH/rtworks`

**OpenVMS:**

`RTHOME:[INCLUDE.RTWORKS]`

**Windows:**

`%RTHOME%\include\rtworks`

Header files should be included with the angle-bracket syntax. For example, when using C, this inclusion line is used:

```
#include <rtworks/ipc.h>
```

While for the C++ class library this inclusion line is used (in fact, the cxxipc.hxx file indirectly includes `ipc.h`):

```
#include <rtworks/cxxipc.hxx>
```

Use of the angle-bracket syntax is preferred over the use of double-quotes:

```
#include "ipc.h"
```

or

#`include "`*pathname*`/ipc.h"`

On UNIX, using angle brackets requires using an argument to the `C++` command:

```
$ CC -I$RTHOME/include/$RTARCH ...
```

On OpenVMS, the logical `RTWORKS` directory is `RTHOME:[INCLUDE.RTWORKS]` so that DEC C++ uses that directory for `<rtworks/cxxipc.hxx>`.

On Windows, add the SmartSockets include directory `%RTHOME%\include\rtworks` to the list of Visual C++ Developer Studio project include directories. There are several example Visual C++ makefiles in `%RTHOME%\examples\cxxipc` that you can use as a starting point for your application executables.

# Utility Functions and Macros

The SmartSockets utility functions (those prefixed with Tut) and utility macros (those prefixed with T_) can be used at any time from either C or C++. For further details, see the *TIBCO SmartSockets Utilities* reference.

# Compiling and Linking

After writing a user-defined program, use the `rtlink` shell script to link it with the necessary SmartSockets libraries. The following sections present some examples of using the `rtlink` script to link the program stored in files `myprog.cxx` and `extfunc.cxx`.

## On UNIX

These examples represent two methods of invoking the `rtlink` script on UNIX. In the first example, the compiling and linking steps are executed separately. In the second, they are combined into one step. Both examples use the `-o` flag to explicitly name the executable, and the `-O` flag to enable compiler and SmartSockets optimization. `rtlink` by default uses the `cc` command to compile and link, but you can override this either by using the `CC` environment variable (see Using a Different Compiler on page 78) or by using the `rtlink -cxx` flag. Specifying the `-cxx` flag tells `rtlink` to use the native C++ compiler (for example, on a Solaris platform the compiler is named `CC`, on AIX `xlC`, on Digital UNIX `cxx`, and so on) and adds the SmartSockets C++ class library to the list of libraries to be linked into the executable. For more information on compiling and linking, see the detailed description of `rtlink` in the *TIBCO SmartSockets Utilities* reference.

```
// Separate compiles followed by a separate link
$ CC -O -I$RTHOME/include/$RTARCH -c myprog.cxx
$ CC -O -I$RTHOME/include/$RTARCH -c extfunc.cxx
$ rtlink -cxx -O myprog.o extfunc.o -o myprog.x

// Combined compiles and link
$ rtlink -cxx -O myprog.cxx extfunc.cxx -o myprog.x
```

### Using a Different Compiler

By default, `rtlink` uses the `cc` command to compile and link files. If the `-cxx` flag is used, `rtlink` uses a native C++ compiler to compile and link files. If the `CC` environment variable is set, `rtlink` uses it to determine the command to invoke. To use another compiler, set the environment variable to the compiler you wish to use. This example uses the GNU C++ compiler:

```
$ env CC=g++ rtlink -O myprog.C extfunc.C -o myprog.x
```

Do not set the `CC` environment variable to `rtlink` because this setting causes `rtlink` to call itself over and over.

### Using Make

If you are using the `make` utility to control and execute your compilations, you can instead set the `CC` variable in your `makefile` and `make` automatically sets the `CC` environment variable each time it runs a command. Here is an example of setting `CC` from a `makefile` to use the GNU C++ compiler:

```
CC = g++
CFLAGS = -O -I$(RTHOME)/include/$(RTARCH)
OBJECTS = myprog.o extfunc.o

myprog.x: $(OBJECTS)
        rtlink -cxx $(OBJECTS) -o $@
```

If you are using GNU make, this may be needed instead in your `makefile` or `GNUmakefile`:

```
export CC = g++
CFLAGS = -O -I$(RTHOME)/include/$(RTARCH)
OBJECTS = myprog.o extfunc.o

myprog.x: $(OBJECTS)
        rtlink -cxx $(OBJECTS) -o $@
```

## On OpenVMS

On OpenVMS, the `rtlink` syntax is similar to UNIX, except you cannot combine the compilation and link steps. These steps must be explicitly invoked. The next example uses the `/executable` qualifier to explicitly name the executable. For more information on compiling and linking, see the detailed description of `rtlink` in the *TIBCO SmartSockets Utilities* reference.

```
// Separate compiles followed by a separate link
$ cxx/optimize myprog.c
$ cxx/optimize extfunc.c
$ rtlink -cxx /executable=myprog.exe myprog.obj extfunc.obj
```

The compilations create object files named `myprog.obj` and `extfunc.obj`. `rtlink` creates an executable named `myprog.exe`.

### On Windows

On Windows, compiling and linking is usually accomplished within the Visual C++ Developer Studio environment. There is no `rtlink` script on Windows. There are several example Visual C++ makefiles in `%RTHOME%\examples\cxxipc` that you can use as a starting point for your application executables.

## Error Handling

SmartSockets provides some simple error handling for its C API functions. The error handling system provides additional information about why a function call failed. This is most often found in functions that return a `TRUE` or `FALSE` status. For function calls that support error handling, the global SmartSockets error number is set to a value that provides additional information about why the function failed. This global error number, which is stored in a per-thread value for multi-threaded programs, can be retrieved with TutErrNumGet and set with TutErrNumSet. To simplify logging or printing notification of errors, a string describing the error can be retrieved with TutErrStrGet or TutErrNumToStr. The Diagnostics section of each function description describes how error handling is supported for that function. The error number should only be used immediately after a function which supports error handling returns a failure value. At any other time, the value of the error number is meaningless.

In the C++ class library, C++ member functions generally do not return a success status, but rather return the result of the corresponding operation. A status flag is maintained in the Tobj base class that can be checked either by using a unary `!` operator or by calling the member function Tobj::Status. Access to the SmartSockets global error number through the TutErrNumGet and TutErrNumSet functions is left unchanged in the C++ class library.

# C++ Class Descriptions

The remaining pages of this chapter describe in detail the syntax and functionality of each of the SmartSockets C++ classes. A reference page is supplied for each class, consisting of:

- **Name** — the name of the class
- **Synopsis** — shows class header file declaration and various class constructors
- **Inheritance** — shows an inheritance tree for the class
- **Description** — describes what the class does
- **Caution** — describes possible side effects
- **See Also** — reference to related classes
- **Example** — shows at least one example of using the class

The classes are listed in alphabetical order.

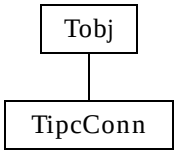
# TipcConn

**Name** TipcConn — user class for a dummy connection and base class for all other connection classes

**Synopsis**

```
TipcConn conn;
TipcConn conn(connection);
TipcConn conn(connection, destroy_flag);
```

**Inheritance**




**Description** The TipcConn class calls the C API function TipcConnCreate at construction. It manages the resulting T_IPC_CONN object internally.

**Caution** None

**Derived Classes** TipcConnServer, TipcConnClient

**See Also** Tobj; see the *TIBCO SmartSockets Application Programming Interface* for information on TipcConnCreate

## Construction/Destruction

### TipcConn::TipcConn

**Syntax:**

```
TipcConn();
```

**Remarks:** Create a TipcConn object.

**Syntax:**

```
TipcConn(T_IPC_CONN connection,
         T_BOOL destroy_flag = TRUE);
```

**Remarks:** Create a TipcConn object with T_IPC_CONN. Optionally set the *destroy_flag* that controls calling the C API function TipcConnDestroy at object destruction. The *destroy_flag* parameter defaults to TRUE. This means that TipcConnDestroy is called when the object is destroyed.

## Protected Data

### TipcConn::_connection

**Syntax:** `T_IPC_CONN _connection;`

**Remarks:** `_connection` is managed on behalf of derived classes. The setting of `_connection` is accomplished by the derived class constructor. The class destructor uses `_connection` when it calls TipcConnDestroy.

### TipcConn::_destroy_flag

**Syntax:** `T_BOOL _destroy_flag;`

**Remarks:** `_destroy_flag` is a flag used by derived classes to control the use of TipcConnDestroy in the class destructor.

### TipcConn::_msg_referent

**Syntax:** `TipcMsg _msg_referent;`

**Remarks:** `_msg_referent` is a holding place for TipcMsg objects returned by reference by member functions of this class.

## Type Conversion Operators

### TipcConn::operator T_IPC_CONN

**Syntax:** `operator T_IPC_CONN();`

**Remarks:** Converts a TipcConn to the C API T_IPC_CONN data type.

## Member Functions

### TipcConn::Arch

**Syntax:** `virtual T_STR Arch();`

**Remarks:** Get the architecture name of a connection's peer process.

**C API:** TipcConnGetArch

### TipcConn::AutoFlushSize

**Syntax:** `virtual T_INT4 AutoFlushSize();`

**Remarks:** Get the automatic flush size of a connection.

**C API:** TipcConnGetAutoFlushSize

**Syntax:** `virtual T_BOOL AutoFlushSize(T_INT4 auto_flush_size);`

**Remarks:** Set the automatic flush size of a connection.

**C API:** TipcConnSetAutoFlushSize

### TipcConn::BlockMode

**Syntax:** `virtual T_BOOL BlockMode();`

**Remarks:** Get the block mode of a connection.

**C API:** TipcConnGetBlockMode

**Syntax:** `virtual T_BOOL BlockMode(T_BOOL block_mode);`

**Remarks:** Set the block mode of a connection.

**C API:** TipcConnSetBlockMode

### TipcConn::Check

**Syntax:**
```
virtual T_BOOL Check(T_IO_CHECK_MODE check_mode,
                     T_REAL8 timeout);
```

**Remarks:** Check if data can be read from or written to a connection.

**C API:** TipcConnCheck

### TipcConn::Connection

**Syntax:** `T_IPC_CONN Connection();`

**Remarks:** Return the C API T_IPC_CONN structure stored in the protected connection.

**C API:** None

### TipcConn::DecodeCbCreate

**Syntax:**
```
virtual T_CB
DecodeCbCreate(T_IPC_CONN_DECODE_CB_FUNC func,
               T_CB_ARG arg);
```

**Remarks:** Create a decode callback in a connection.

**C API:** TipcConnDecodeCbCreate

### TipcConn::DecodeCbLookup

**Syntax:**
```
virtual T_CB
DecodeCbLookup(T_IPC_CONN_DECODE_CB_FUNC func,
               T_CB_ARG arg);
```

**Remarks:** Look up a decode callback in a connection.

**C API:** TipcConnDecodeCbLookup

### TipcConn::DefaultCbCreate

**Syntax:**
```
virtual T_CB
DefaultCbCreate(T_IPC_CONN_DEFAULT_CB_FUNC func,
                T_CB_ARG arg);
```

**Remarks:** Create a default callback in a connection.

**C API:** TipcConnDefaultCbCreate

### TipcConn::DefaultCbLookup

**Syntax:**
```
virtual T_CB
DefaultCbLookup(T_IPC_CONN_DEFAULT_CB_FUNC func,
                T_CB_ARG arg);
```

**Remarks:** Look up a default callback in a connection.

**C API:** TipcConnDefaultCbLookup

### TipcConn::EncodeCbCreate

**Syntax:**
```
virtual T_CB
EncodeCbCreate(T_IPC_CONN_ENCODE_CB_FUNC func,
               T_CB_ARG arg);
```

**Remarks:** Create an encode callback in a connection.

**C API:** TipcConnEncodeCbCreate

### TipcConn::EncodeCbLookup

**Syntax:**
```
virtual T_CB
EncodeCbLookup(T_IPC_CONN_ENCODE_CB_FUNC func,
               T_CB_ARG arg);
```

**Remarks:** Look up an encode callback in a connection.

**C API:** TipcConnEncodeCbLookup

### TipcConn::Arch

**Syntax:**
```
virtual T_STR Arch();
```

**Remarks:** Get the architecture name of a connection's peer process.

**C API:** TipcConnGetArch

### TipcConn::ErrorCbCreate

**Syntax:**
```
virtual T_CB ErrorCbCreate(T_IPC_CONN_ERROR_CB_FUNC func,
                           T_CB_ARG arg);
```

**Remarks:** Create an error callback in a connection.

**C API:** TipcConnErrorCbCreate

### TipcConn::ErrorCbLookup

**Syntax:**
```
virtual T_CB ErrorCbLookup(T_IPC_CONN_ERROR_CB_FUNC func,
                           T_CB_ARG arg);
```

**Remarks:** Look up an error callback in a connection.

**C API:** TipcConnErrorCbLookup

### TipcConn::Flush

**Syntax:** `virtual T_BOOL Flush();`

**Remarks:** Flush buffered outgoing messages to connection's socket.

**C API:** TipcConnFlush

### TipcConn::GmdFileCreate

**Syntax:** `virtual T_BOOL GmdFileCreate();`

**Remarks:** Create guaranteed message delivery area on a connection.

**C API:** TipcConnGmdFileCreate

### TipcConn::GmdFileDelete

**Syntax:** `virtual T_BOOL GmdFileDelete();`

**Remarks:** Delete guaranteed message delivery files for a connection.

**C API:** TipcConnGmdFileDelete

### TipcConn::GmdMaxSize

**Syntax:** `virtual T_UINT4 GmdMaxSize();`

**Remarks:** Get the GMD area maximum size of a connection.

**C API:** TipcConnGetGmdMaxSize

**Syntax:** `virtual T_BOOL GmdMaxSize(T_UINT4 gmd_max_size);`

**Remarks:** Set the GMD area maximum size of a connection.

**C API:** `TipcConnSetGmdMaxSize`

### TipcConn::GmdMsgDelete

**Syntax:** `virtual T_BOOL GmdMsgDelete(TipcMsg& msg);`

**Remarks:** Delete a message from GMD area after a GMD failure on a connection.

**C API:** TipcConnGmdMsgDelete

### TipcConn::GmdMsgResend

**Syntax:** `virtual T_BOOL GmdMsgResend(TipcMsg& msg);`

**Remarks:** Resend a message after a GMD failure on a connection.

**C API:** TipcConnGmdMsgResend

### TipcConn::GmdNumPending

**Syntax:** `virtual T_INT4 GmdNumPending();`

**Remarks:** Get the number of outgoing guaranteed messages pending.

**C API:** TipcConnGetGmdNumPending

### TipcConn::GmdResend

**Syntax:** `virtual T_BOOL GmdResend();`

**Remarks:** Resend all guaranteed messages after a delivery failure on a connection.

**C API:** TipcConnGmdResend

### TipcConn::Insert

**Syntax:** `virtual T_BOOL Insert(TipcMsg& msg, T_INT4 pos);`

**Remarks:** Insert a message into the message queue of a connection.

**C API:** TipcConnMsgInsert

### TipcConn::KeepAlive

**Syntax:** `virtual T_BOOL KeepAlive();`

**Remarks:** Check if the process at the other end of a connection is still alive.

**C API:** TipcConnKeepAlive

### TipcConn::Lock

**Syntax:** `virtual T_BOOL Lock();`

**Remarks:** Acquire exclusive access to a connection.

**C API:** TipcConnLock

### TipcConn::MainLoop

**Syntax:** `virtual T_BOOL MainLoop(T_REAL8 timeout);`

**Remarks:** Read and process messages on a connection.

**C API:** TipcConnMainLoop

### TipcConn::Next

**Syntax:** `virtual TipcMsg& Next(T_REAL8 timeout);`

**Remarks:** Get the next message from a connection.

**C API:** TipcConnMsgNext

### TipcConn::Node

**Syntax:** `virtual T_STR Node();`

**Remarks:** Get the node name of a connection's peer process.

**C API:** TipcConnGetNode

### TipcConn::NumQueued

**Syntax:** `virtual T_INT4 NumQueued();`

**Remarks:** Get number of queued messages from a connection.

**C API:** TipcConnGetNumQueued

### TipcConn::Pid

**Syntax:** `virtual Pid();`

**Remarks:** Get the process ID of a connection's peer process.

**C API:** TipcConnGetPid

### TipcConn::Process

**Syntax:** `virtual T_BOOL Process(TipcMsg& msg);`

**Remarks:** Process a message in a connection.

**C API:** TipcConnMsgProcess

### TipcConn::ProcessCbCreate

**Syntax:**
```
virtual T_CB
ProcessCbCreate(TipcMt& mt,
                T_IPC_CONN_PROCESS_CB_FUNC func,
                T_CB_ARG arg);
```

**Remarks:** Create a process callback in a connection.

**C API:** TipcConnProcessCbCreate

### TipcConn::ProcessCbLookup

**Syntax:**
```
virtual T_CB
ProcessCbLookup(TipcMt& mt,
                T_IPC_CONN_PROCESS_CB_FUNC func,
                T_CB_ARG arg);
```

**Remarks:** Look up a process callback in a connection.

**C API:** TipcConnProcessCbLookup

### TipcConn::QueueCbCreate

**Syntax:**
```
virtual T_CB QueueCbCreate(TipcMt& mt,
                           T_IPC_CONN_QUEUE_CB_FUNC func,
                           T_CB_ARG arg);
```

**Remarks:** Create a queue callback in a connection.

**C API:** TipcConnQueueCbCreate

### TipcConn::QueueCbLookup

**Syntax:**
```
virtual T_CB QueueCbLookup(TipcMt& mt,
                           T_IPC_CONN_QUEUE_CB_FUNC func,
                           T_CB_ARG arg);
```

**Remarks:** Look up a queue callback in a connection.

**C API:** TipcConnQueueCbLookup

### TipcConn::Read

**Syntax:**
```
virtual T_BOOL Read(T_REAL8 timeout);
```

**Remarks:** Read all available data from a connection and queue messages in priority order.

**C API:** TipcConnRead

### TipcConn::ReadCbCreate

**Syntax:**
```
virtual T_CB
ReadCbCreate(TipcMt& mt, T_IPC_CONN_READ_CB_FUNC func,
             T_CB_ARG arg);
```

**Remarks:** Create a read callback in a connection.

**C API:** TipcConnReadCbCreate

### TipcConn::ReadCbLookup

**Syntax:**
```
virtual T_CB
ReadCbLookup(TipcMt& mt, T_IPC_CONN_READ_CB_FUNC func,
             T_CB_ARG arg);
```

**Remarks:** Look up a read callback in a connection.

**C API:** TipcConnReadCbLookup

### TipcConn::Search

**Syntax:**
```
virtual TipcMsg&
Search(T_REAL8 timeout, T_IPC_CONN_MSG_SEARCH_FUNC func,
       T_PTR arg);
```

**Remarks:** Search the message queue of a connection for a specific message.

**C API:** TipcConnMsgSearch

### TipcConn::SearchType

**Syntax:**
```
virtual TipcMsg&(T_REAL8 timeout, TipcMt& mt);
```

**Remarks:** Search the message queue of a connection for a message with a specific type.

**C API:** TipcConnMsgSearchType

### TipcConn::Send

**Syntax:**
```
virtual T_BOOL Send(TipcMsg& msg,
                    T_BOOL check_server_msg_send = FALSE);
```

**Remarks:** Send a message through a connection. For peer-to-peer connections, the *check_server_msg_send* flag is ignored. The TipcSrv derived class is the only connection class that uses the *check_server_msg_send* parameter.

**C API:** TipcConnMsgSend

### TipcConn::SendRpc

**Syntax:** `virtual TipcMsg& SendRpc(TipcMsg& msg, T_REAL8 timeout);`

**Remarks:** Make a remote procedure call (RPC) with messages on a connection.

**C API:** TipcConnMsgSendRpc

### TipcConn::Socket

**Syntax:** `virtual T_INT4 Socket();`

**Remarks:** Get the socket of a connection.

**C API:** TipcConnGetSocket

**Syntax:** `virtual T_BOOL Socket(T_INT4 socket);`

**Remarks:** Set the socket of a connection.

**C API:** TipcConnSetSocket

### TipcConn::Timeout

**Syntax:** `virtual T_REAL8 Timeout(T_IPC_TIMEOUT timeout);`

**Remarks:** Get a timeout property of a connection.

**C API:** TipcConnGetTimeout

**Syntax:** `virtual T_BOOL Timeout(T_IPC_TIMEOUT timeout, T_REAL8 value);`

**Remarks:** Set a timeout property of a connection.

**C API:** TipcConnSetTimeout

### TipcConn::UniqueSubject

**Syntax:** `virtual T_STR UniqueSubject();`

**Remarks:** Get the unique subject of a connection's peer process.

**C API:** TipcConnGetUniqueSubject

### TipcConn::Unlock

**Syntax:** `virtual T_BOOL Unlock();`

**Remarks:** Release exclusive access to a connection.

**C API:** TipcConnUnlock

### TipcConn::User

**Syntax:** `virtual T_STR User();`

**Remarks:** Get the user name of a connection's peer process.

**C API:** TipcConnGetUser

### TipcConn::Write

**Syntax:** `virtual T_BOOL Write(TipcMt& mt, ...);`

**Remarks:** Construct a message and send it through a connection.

**C API:** TipcConnMsgWrite

### TipcConn::WriteCbCreate

**Syntax:**
```
virtual T_CB WriteCbCreate(TipcMt& mt,
                           T_IPC_CONN_WRITE_CB_FUNC func,
                           T_CB_ARG arg);
```

**Remarks:** Create a write callback in a connection.

**C API:** TipcConnWriteCbCreate

### TipcConn::WriteCbLookup

**Syntax:**
```
virtual T_CB WriteCbLookup(TipcMt& mt,
                           T_IPC_CONN_WRITE_CB_FUNC func,
                           T_CB_ARG arg);
```

**Remarks:** Look up a write callback in a connection.

**C API:** TipcConnWriteCbLookup

### TipcConn::WriteVa

**Syntax:** `virtual T_BOOL WriteVa(TipcMt& mt, va_list var_arg_list);`

**Remarks:** Construct a message and send it through a connection.

**C API:** TipcConnMsgWriteVa

### TipcConn::XtSource

**Syntax:** `virtual T_INT4 XtSource();`

**Remarks:** Get source suitable for XtAppAddInput from a connection.

**C API:** TipcConnGetXtSource

## Static Public Member Functions

### TipcConn::GmdDir

**Syntax:** `static T_STR GmdDir();`

**Remarks:** Get name of directory where files are written for guaranteed message delivery.

**C API:** TipcGetGmdDir

## Example

This code fragment creates a connection, creates process and default callbacks, then reads and processes messages from the message file `data.msg`:

```
// Construct a TipcConn
TipcConn my_conn;

// Create a process callback for TIME messages
TipcMt mt_time(T_MT_TIME);
if (!mt_time) {
 // error
}

my_conn.ProcessCbCreate(mt_time, process_time, NULL);
if (!my_conn) {
 // error
}

// Create a process callback for NUMERIC_DATA messages
TipcMt mt_numeric_data(T_MT_NUMERIC_DATA);
if (!mt_numeric_data) {
 // error
}

my_conn.ProcessCbCreate(mt_numeric_data, process_nd, NULL);
if (!my_conn) {
 // error
}

// Create a default callback for all other messages
my_conn.DefaultCbCreate(process_default, NULL);
if (!my_conn) {
 // error
}
```

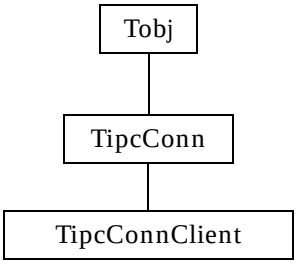
```
// Create a message file that is open for reading
TipcMsgFile msg_file("data.msg", T_IPC_MSG_FILE_CREATE_READ);
if (!msg_file) {
 // error
}

// Read and process messages

TipcMsg msg_read;
while (1) {
  msg_file >> msg_read;
  if (!msg_file) {
    break;
  }
  my_conn.Process(msg_read);
  if (!my_conn) {
    // error
  }

 // Explicitly destroy the message. This step could be skipped
 // since the extraction operator will first destroy any message
 // that msg_read is managing before assigning a new message to it.

 msg_read.Destroy();
}
```

# TipcConnClient

**Name** TipcConnClient — user class for a client connection

**Synopsis** `TipcConnClient conn(`*logical_conn_name*`);`

**Inheritance**




**Description** The TipcConnClient class calls the C API function TipcConnClientCreate at construction, passing the resulting T_IPC_CONN handle up to the TipcConn base class, where it is managed.

**Caution** None

**See Also** Tobj; see the *TIBCO SmartSockets Application Programming Interface* for information on TipcConnCreate and TipcConnClientCreate

## Construction/Destruction

### TipcConnClient::TipcConnClient

**Syntax:** `TipcConnClient(T_STR` *logical_conn_name*`);`

**Remarks:** Create a TipcConnClient object with *logical_conn_name*.

**C API:** TipcConnCreateClient

## Example

This code fragment creates a client connection, then sends an INFO message to the other side:

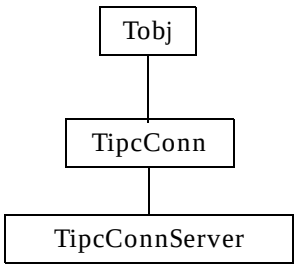
```
TipcConn *conn = new TipcConnClient("tcp:_node:5252");
if (!conn) {
  // error
}
TipcMsg msg(T_MT_INFO);
msg.AppendStr("hiya");
conn->Send(msg);
```

# TipcConnServer

**Name** TipcConnServer — user class for a server connection

**Synopsis** `TipcConnServer conn(`*logical_conn_name*`);`

**Inheritance**




**Description** The TipcConnServer class calls the C API function TipcConnServerCreate at construction, passing the resulting T_IPC_CONN handle up to the TipcConn base class, where it is managed.

**Caution** None

**See Also** Tobj; see the *TIBCO SmartSockets Application Programming Interface* for information on TipcConnCreate and TipcConnServerCreate

## Construction/Destruction

### TipcConnServer::TipcConnServer

**Syntax:** `TipcConnServer(T_STR` *logical_conn_name*`);`

**Remarks:** Create a TipcConnServer object with *logical_conn_name.*

**C API:** TipcConnCreateServer

## Public Member Function

### TipcConnServer::Accept

**Syntax:** `TipcConn *Accept();`

**Remarks:** Accepts a connection from a client and returns a pointer to a new TipcConn object. It is your responsibility to delete this object when finished.

**C API:** TipcConnAccept

## Example

This code fragment creates a TCP server connection using port `5252` on the node named `moose`, waits for a client to connect, accepts that connection, and reads a packet from the client connection:

```
TipcConnServer *srv_conn = new TipcConnServer("tcp:moose:5252");
if (!srv_conn) {
  // error
}

TipcConn *conn = srv_conn->Accept();
if (!conn) {
  // error
}

TipcMsg msg;

msg = conn->Next(10.0);
```

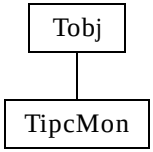
# TipcMon

**Name** TipcMon — user class for monitoring RTserver and the RTclients attached to RTserver

**Synopsis**

```
TipcMon mon;
```

**Inheritance**




**Description** A TipcMon object provides methods for an RTclient to monitor information that is not associated with a particular RTserver or RTclient.

**Caution** None

**See Also** TipcMonClient, TipcMonServer

## Construction/Destruction

### TipcMon::TipcMon

**Syntax:**

```
TipcMon();
```

**Remarks:** Create a TipcMon object.

## Public Member Functions

### TipcMon::ClientNamesPoll

**Syntax:** `T_BOOL ClientNamesPoll();`

**Remarks:** Poll once for RTclient names.

**C API:** TipcMonClientNamesPoll

### TipcMon::ClientNamesWatch

**Syntax:** `T_BOOL ClientNamesWatch();`

**Remarks:** Get whether or not this RTclient is watching RTclient names.

**C API:** TipcMonClientNamesGetWatch

**Syntax:** `T_BOOL ClientNamesWatch(T_BOOL` *watch_status*`);`

**Remarks:** Start or stop watching RTclient names.

**C API:** TipcMonClientNamesSetWatch

### TipcMon::IdentStr

**Syntax:** `T_STR IdentStr();`

**Remarks:** Get the monitoring identification string of this process.

**C API:** TipcMonGetIdentStr

**Syntax:** `T_BOOL IdentStr(T_STR` *type_str*`);`

**Remarks:** Set the monitoring identification string of this process.

**C API:** TipcMonSetIdentStr

### TipcMon::PrintWatch

**Syntax:** `T_BOOL PrintWatch(T_OUT_FUNC` *out_func*`);`

**Remarks:** Print all monitoring categories being watched.

**C API:** TipcMonPrintWatch

### TipcMon::ProjectNamesPoll

**Syntax:** `T_BOOL ProjectNamesPoll();`

**Remarks:** Poll once for project names.

**C API:** TipcMonProjectNamesPoll

### TipcMon::ProjectNamesWatch

**Syntax:** `T_BOOL ProjectNamesWatch();`

**Remarks:** Get whether or not this RTclient is watching project names.

**C API:** TipcMonProjectNamesGetWatch

**Syntax:** `T_BOOL ProjectNamesWatch(T_BOOL` *watch_status*`);`

**Remarks:** Start or stop watching project names.

**C API:** TipcMonProjectNamesSetWatch

### TipcMon::ServerConnPoll

**Syntax:** `T_BOOL ServerConnPoll();`

**Remarks:** Poll once for RTserver connections.

**C API:** TipcMonServerConnPoll

### TipcMon::ServerConnWatch

**Syntax:** `T_BOOL ServerConnWatch();`

**Remarks:** Get whether or not this RTclient is watching RTserver connections.

**C API:** TipcMonServerConnGetWatch

**Syntax:** `T_BOOL ServerConnWatch(T_BOOL` *watch_status*`);`

**Remarks:** Start or stop watching RTserver connections.

**C API:** TipcMonServerConnSetWatch

### TipcMon::ServerNamesPoll

**Syntax:** `T_BOOL ServerNamesPoll();`

**Remarks:** Poll once for RTserver names.

**C API:** TipcMonServerNamesPoll

### TipcMon::ServerNamesWatch

**Syntax:** `T_BOOL ServerNamesWatch();`

**Remarks:** Get whether or not this RTclient is watching RTserver names.

**C API:** TipcMonServerNamesGetWatch

**Syntax:** `T_BOOL ServerNamesWatch(T_BOOL` *watch_status*`);`

**Remarks:** Start or stop watching RTserver names.

**C API:** TipcMonServerNamesSetWatch

### TipcMon::SubjectNamesPoll

**Syntax:** `T_BOOL SubjectNamesPoll();`

**Remarks:** Poll once for subject names.

**C API:** TipcMonSubjectNamesPoll

### TipcMon::SubjectNamesWatch

**Syntax:** `T_BOOL SubjectNamesWatch();`

**Remarks:** Get whether or not this RTclient is watching subject names.

**C API:** TipcMonSubjectNamesGetWatch

**Syntax:** `T_BOOL SubjectNamesWatch(T_BOOL` *watch_status*`);`

**Remarks:** Start or stop watching subject names.

**C API:** TipcMonSubjectNamesSetWatch

### TipcMon::SubjectSubscribePoll

**Syntax:** `T_BOOL SubjectSubscribePoll(T_STR` *subject_name*`);`

**Remarks:** Poll once for the names of the RTclients that are subscribing to a subject.

**C API:** TipcMonSubjectSubscribePoll

### TipcMon::SubjectSubscribeWatch

**Syntax:** `T_BOOL SubjectSubscribeWatch(T_STR` *subject_name*`);`

**Remarks:** Get whether or not this RTclient is watching the RTclients that are subscribing to a subject.

**C API:** TipcMonSubjectSubscribeGetWatch

**Syntax:** `T_BOOL SubjectSubscribeWatch(T_STR` *subject_name*`,`
`T_BOOL` *watch_status*`);`

**Remarks:** Start or stop watching RTclients that are subscribing to a subject.

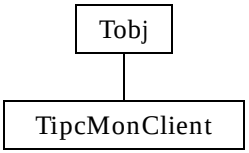
**C API:** TipcMonSubjectSubscribeSetWatch

# TipcMonClient

**Name** TipcMonClient — user class for monitoring a specific RTclient

**Synopsis** `TipcMonClient mon(`*client_name*`);`

**Inheritance**




**Description** A TipcMonClient object provides methods for an RTclient to monitor information for a particular RTclient.

**Caution** None

**See Also** TipcMon, TipcMonServer

## Construction/Destruction

### TipcMonClient::TipcMonClient

**Syntax:** `TipcMonClient(T_STR` *client_name*`);`

**Remarks:** Create a TipcMonClient object using *client_name*. *client_name* is the name of an RTclient to be monitored.

## Public Member Functions

### TipcMonClient::BufferPoll

**Syntax:** `T_BOOL BufferPoll();`

**Remarks:** Poll once for RTclient message-related buffer information.

**C API:** TipcMonClientBufferPoll

### TipcMonClient::BufferWatch

**Syntax:** `T_BOOL BufferWatch();`

**Remarks:** Get whether or not this RTclient is watching message-related buffer information in an RTclient.

**C API:** TipcMonClientBufferGetWatch

**Syntax:** `T_BOOL BufferWatch(T_BOOL` *watch_status*`);`

**Remarks:** Start or stop watching RTclient message-related buffer information.

**C API:** TipcMonClientBufferSetWatch

### TipcMonClient::CbPoll

**Syntax:** `T_BOOL CbPoll();`

**Remarks:** Poll once for RTclient callback information.

**C API:** TipcMonClientCbPoll

### TipcMonClient::GeneralPoll

**Syntax:** `T_BOOL GeneralPoll();`

**Remarks:** Poll once for RTclient general information.

**C API:** TipcMonClientGeneralPoll

### TipcMonClient::MsgRecvWatch

**Syntax:** `T_BOOL MsgRecvWatch(T_STR` *msg_type_name*`);`

**Remarks:** Get whether or not this RTclient is watching received messages in an RTclient.

**C API:** TipcMonClientMsgRecvGetWatch

**Syntax:** `T_BOOL MsgRecvWatch(T_STR` *msg_type_name*`, T_BOOL` *watch_status*`);`

**Remarks:** Start or stop watching RTclient received messages.

**C API:** TipcMonClientMsgRecvSetWatch

### TipcMonClient::MsgSendWatch

**Syntax:** `T_BOOL MsgSendWatch(T_STR msg_type_name);`

**Remarks:** Get whether or not this RTclient is watching sent messages in an RTclient.

**C API:** TipcMonClientMsgSendGetWatch

**Syntax:** `T_BOOL MsgSendWatch(T_STR msg_type_name, T_BOOL watch_status);`

**Remarks:** Start or stop watching RTclient sent messages.

**C API:** TipcMonClientMsgSendSetWatch

### TipcMonClient::MsgTrafficPoll

**Syntax:** `T_BOOL MsgTrafficPoll();`

**Remarks:** Poll once for RTclient message traffic information.

**C API:** TipcMonClientMsgTrafficPoll

### TipcMonClient::MsgTypePoll

**Syntax:** `T_BOOL MsgTypePoll(T_STR msg_type_name);`

**Remarks:** Poll once for RTclient message type information.

**C API:** TipcMonClientMsgTypePoll

### TipcMonClient::OptionPoll

**Syntax:** `T_BOOL OptionPoll(T_STR option_name);`

**Remarks:** Poll once for RTclient option information.

**C API:** TipcMonClientOptionPoll

### TipcMonClient::SubscribePoll

**Syntax:** `T_BOOL SubscribePoll();`

**Remarks:** Poll once for the names of the subjects to which an RTclient is subscribing.

**C API:** TipcMonClientSubscribePoll

### TipcMonClient::SubscribeWatch

**Syntax:** `T_BOOL SubscribeWatch();`

**Remarks:** Get whether or not this RTclient is watching the subjects to which an RTclient is subscribing.

**C API:** TipcMonClientSubscribeGetWatch

**Syntax:** `T_BOOL SubscribeWatch(T_BOOL` *watch_status*`);`

**Remarks:** Start or stop watching the subjects to which an RTclient is subscribing.

**C API:** TipcMonClientSubscribeSetWatch

### TipcMonClient::SubjectPoll

**Syntax:** `T_BOOL SubjectPoll(T_STR` *subject_name*`);`

**Remarks:** Poll once for RTclient subject information.

**C API:** TipcMonClientSubjectPoll

### TipcMonClient::TimePoll

**Syntax:** `T_BOOL TimePoll();`

**Remarks:** Poll once for RTclient time information.

**C API:** TipcMonClientTimePoll

### TipcMonClient::TimeWatch

**Syntax:** `T_BOOL TimeWatch();`

**Remarks:** Get whether or not this RTclient is watching time information.

**C API:** TipcMonClientTimeGetWatch

**Syntax:** `T_BOOL TimeWatch(T_BOOL` *watch_status*`);`

**Remarks:** Start or stop watching RTclient time information.

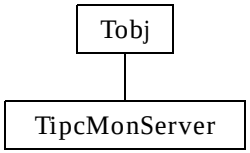
**C API:** TipcMonClientTimeSetWatch

# TipcMonServer

**Name** TipcMonServer — user class for monitoring a specific RTserver

**Synopsis** `TipcMonServer mon(`*server_name*`);`

**Inheritance**




**Description** A TipcMonServer object provides methods for an RTclient to monitor information for a particular RTserver.

**Caution** None

**See Also** TipcMon, TipcMonClient

## Construction/Destruction

### TipcMonServer::TipcMonServer

**Syntax:** `TipcMonServer(T_STR` *server_name*`);`

**Remarks:** Create a TipcMonServer object using *server_name*. *server_name* is the name of an RTserver to be monitored.

## Public Member Functions

### TipcMonServer::BufferPoll

**Syntax:** `T_BOOL BufferPoll(T_STR` *connected_process_name*`);`

**Remarks:** Poll once for RTserver message-related buffer information.

**C API:** TipcMonServerBufferPoll

### TipcMonServer::GeneralPoll

**Syntax:** `T_BOOL GeneralPoll();`

**Remarks:** Poll once for RTserver general information.

**C API:** TipcMonServerGeneralPoll

### TipcMonServer::MsgTrafficPoll

**Syntax:** `T_BOOL MsgTrafficPoll(T_STR` *connected_process_name*`);`

**Remarks:** Poll once for RTserver message traffic information.

**C API:** TipcMonServerMsgTrafficPoll

### TipcMonServer::OptionPoll

**Syntax:** `T_BOOL OptionPoll(T_STR` *option_name*`);`

**Remarks:** Poll once for RTserver option information.

**C API:** TipcMonServerOptionPoll

### TipcMonServer::RoutePoll

**Syntax:** `T_BOOL RoutePoll(T_STR` *dest_server_name*`);`

**Remarks:** Poll once for RTserver route information.

**C API:** TipcMonServerRoutePoll

### TipcMonServer::TimePoll

**Syntax:** `T_BOOL TimePoll();`

**Remarks:** Poll once for RTserver time information.

**C API:** TipcMonServerTimePoll

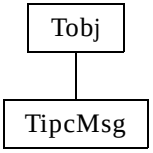
# TipcMsg

**Name** TipcMsg — user class for constructing, manipulating, and eventually destroying messages

**Synopsis**

```
TipcMsg msg;
TipcMsg msg(mt_obj);
TipcMsg msg(mt_num);
TipcMsg msg(mt_num, destination);
TipcMsg msg(mt_num, destination, sender);
TipcMsg msg(mt_name);
TipcMsg msg(c_api_msg);
TipcMsg msg(msg_object);
```

**Inheritance**




**Description** The TipcMsg class acts as a C++ interface to the SmartSockets TipcMsg* C language API functions. The methods of the TipcMsg class allow you to construct messages and to manipulate the data of a message. See the *TIBCO SmartSockets User's Guide* for more information on message composition in SmartSockets applications.

The TipcMsg class has two ways of appending and accessing data fields from a message. The first is designed to provide an interface similar to the C API and consists of several overloaded Append and Next member functions. The Append() and Next() member functions replace the prefixed TipcMsgAppend* and TipcMsgNext* functions of the C API. See Using Append and Next Functions on page 30 for examples using insertion and extraction operators.

The second method of appending and accessing data fields is designed to provide an interface similar to C++ IOstream classes. Overloaded insertion (operator<<) and extraction (operator>>) operators replace the prefixed TipcMsgAppend* and TipcMsgNext* functions of the C API. Examples using insertion and extraction operators are presented in Using Append and Next Functions on page 30.

The TipcMsg class contains a private SmartSockets C API T_IPC_MSG message.

**Caution** None

**See Also** Tobj

## Construction/Destruction

### TipcMsg::TipcMsg

**Syntax:** `TipcMsg();`

**Remarks:** Create a vacant TipcMsg object. When this constructor is used, the created object is vacant until it is either assigned a T_IPC_MSG message or a non-empty TipcMsg object with `operator=`.

**Syntax:** `TipcMsg(const TipcMt& mt_obj);`

**Remarks:** Create a TipcMsg object with *mt_obj*. *mt_obj* is a SmartSockets TipcMt object used to create the message contained in a TipcMsg object. The object's internal T_IPC_MSG variable is initialized by a call to TipcMsgCreate().

**Syntax:**
```
TipcMsg(T_INT4 mt_num, T_STR destination = (T_STR)NULL,
        T_STR sender = (T_STR)NULL);
```

**Remarks:** Create a TipcMsg object with *mt_num*, *destination*, and *sender*. *mt_num* is the number of the message type used to create the message contained in a TipcMsg object by a call to TipcMsgCreate(). *destination* is the destination of the message. *sender* is the sender of the message. *sender* and *destination* default to `NULL`, meaning no sender or destination is to be specified, respectively, when the object is constructed.

**Syntax:** `TipcMsg(T_STR mt_name);`

**Remarks:** Create a TipcMsg object with *mt_name*. *mt_name* is the name of a message type (not case sensitive) to be used to create a TipcMsg object. The object's internal T_IPC_MSG variable is initialized by a call to TipcMsgCreate().

**Syntax:** `TipcMsg(const T_IPC_MSG msg);`

**Remarks:** Create a TipcMsg object with *msg*. *msg* is a SmartSockets C API T_IPC_MSG message used in creating a TipcMsg object. The internal T_IPC_MSG variable of the object is assigned the value of *msg* and the reference count of *msg* is incremented by a call to TipcMsgIncrRefCount().

**Syntax:** `TipcMsg(const TipcMsg *msg_obj*);`

**Remarks:** This constructor is a copy constructor used to create a TipcMsg object with another TipcMsg object called *msg_obj*. The internal T_IPC_MSG pointer of the created object is assigned the value of the internal T_IPC_MSG pointer of *msg_obj* and the reference count for that particular T_IPC_MSG message is incremented by calling TipcMsgIncrRefCount().

### TipcMsg::~TipcMsg()

**Syntax:** `~TipcMsg();`

**Remarks:** Destroy a TipcMsg object. The destructor calls TipcMsgDestroy() if the internal T_IPC_MSG variable of the object is not null and not read only. See the detailed description of TipcMsgDestroy in the *TIBCO SmartSockets Application Programming Interface* for more information regarding how the T_IPC_MSG variable's reference count mechanism is used to control when the memory of the T_IPC_MSG is freed.

## Public Member Functions

### TipcMsg::Ack

**Syntax:** `T_BOOL Ack();`

**Remarks:** Acknowledge delivery of a message.

**C API:** TipcMsgAck

### TipcMsg::AddNamed

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_PTR value, T_INT4 *size*);`

**Remarks:** Add a BINARY field to a message using a name.

**C API:** TipcMsgAddNamedBinary

**Syntax:**
```
T_BOOL AddNamed(T_STR name, T_PTR ptr, T_INT4 size,
                T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Add a BINARY field to a message using a name and a pointer to the field value, rather than a copy.

**C API:** TipcMsgAddNamedBinaryPtr

**Syntax:** `T_BOOL AddNamedBool(T_STR *name*, T_BOOL *value*);`

**Remarks:** Add a BOOL field to a message using a name.

**C API:** TipcMsgAddNamedBool

**Syntax:** `T_BOOL AddNamedBool(T_STR *name*, T_BOOL *value, T_INT4 *size*);`

**Remarks:** Add a field containing an array of BOOL fields to a message using a name.

**C API:** TipcMsgAddNamedBoolArray

**Syntax:** `T_BOOL AddNamedBool(T_STR *name*, T_BOOL *value, T_INT4 *size*, T_IPC_MSG_FIELD *field_return*);`

**Remarks:** Add a field containing an array of BOOL fields to a message using a name and a pointer to the field value, rather than a copy.

**C API:** TipcMsgAddNamedBoolArrayPtr

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_UCHAR *value*);`

**Remarks:** Add a BYTE field to a message using a name.

**C API:** TipcMsgAddNamedByte

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_CHAR *value*);`

**Remarks:** Add a CHAR field to a message using a name.

**C API:** TipcMsgAddNamedChar

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_INT2 *value*);`

**Remarks:** Add an INT2 field to a message using a name.

**C API:** TipcMsgAddNamedInt2

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_INT2 *value, T_INT4 *size*);`

**Remarks:** Add a field containing an array of INT2 fields to a message using a name.

**C API:** TipcMsgAddNamedInt2Array

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_INT2 **value*, T_INT4 *size*, T_IPC_MSG_FIELD **field_return*);`

**Remarks:** Add a field containing an array of INT2 fields to a message using a name and a pointer to the field value, rather than a copy.

**C API:** TipcMsgAddNamedInt2ArrayPtr

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_INT4 *value*);`

**Remarks:** Add an INT4 field to a message using a name.

**C API:** TipcMsgAddNamedInt4

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_INT4 **value*, T_INT4 *size*);`

**Remarks:** Add a field containing an array of INT4 fields to a message using a name.

**C API:** TipcMsgAddNamedInt4Array

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_INT4 **value*, T_INT4 *size*, T_IPC_MSG_FIELD **field_return*);`

**Remarks:** Add a field containing an array of INT4 fields to a message using a name and a pointer to the field value, rather than a copy.

**C API:** TipcMsgAddNamedInt4ArrayPtr

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_INT8 *value*);`

**Remarks:** Add an INT8 field to a message using a name.

**C API:** TipcMsgAddNamedInt8

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_INT8 **value*, T_INT4 *size*);`

**Remarks:** Add a field containing an array of INT8 fields to a message using a name.

**C API:** TipcMsgAddNamedInt8Array

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_INT8 **value*, T_INT4 *size*, T_IPC_MSG_FIELD **field_return*);`

**Remarks:** Add a field containing an array of INT8 fields to a message using a name and a pointer to the field value, rather than a copy.

**C API:** TipcMsgAddNamedInt8ArrayPtr

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_IPC_MSG *value*);`

**Remarks:** Add a T_IPC_MSG field to a message using a name.

**C API:** TipcMsgAddNamedMsg

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_PTR *ptr*);`

**Remarks:** Add a MSG field to a message using a name and a pointer to the field value, rather than a copy.

**C API:** TipcMsgAddNamedMsgPtr

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_MSG *value, T_INT4 *size*);`

**Remarks:** Add a field containing an array of MSG fields to a message using a name.

**C API:** TipcMsgAddNamedMsgArray

**Syntax:**
```
T_BOOL AddNamed(T_STR name, T_MSG *value, T_INT4 size,
                T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Add a field containing an array of MSG fields to a message using a name and a pointer to the field value, rather than a copy.

**C API:** TipcMsgAddNamedMsgArrayPtr

**Syntax:**
```
T_BOOL AddNamed(T_STR name, T_PTR ptr, T_INT4 size,
                T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Add a MSG field to a message using a pointer.

**C API:** TipcMsgAddNamedMsgPtr

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_REAL4 *value*);`

**Remarks:** Add a REAL4 field to a message using a name.

**C API:** TipcMsgAddNamedReal4

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_REAL4 *value, T_INT4 *size*);`

**Remarks:** Add a field containing an array of REAL4 fields to a message using a name.

**C API:** TipcMsgAddNamedReal4Array

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_REAL4 **value*, T_INT4 *size*, T_IPC_MSG_FIELD **field_return*);`

**Remarks:** Add a field containing an array of REAL4 fields to a message using a name and a pointer to the field value, rather than a copy.

**C API:** TipcMsgAddNamedReal4ArrayPtr

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_REAL8 *value*);`

**Remarks:** Add a REAL8 field to a message using a name.

**C API:** TipcMsgAddNamedReal8

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_REAL8 **value*, T_INT4 *size*);`

**Remarks:** Add a field containing an array of REAL8 fields to a message using a name.

**C API:** TipcMsgAddNamedReal8Array

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_REAL8 **value*, T_INT4 *size*, T_IPC_MSG_FIELD **field_return*);`

**Remarks:** Add a field containing an array of REAL8 fields to a message using a name and a pointer to the field value, rather than a copy.

**C API:** TipcMsgAddNamedReal8ArrayPtr

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_REAL16 *value*);`

**Remarks:** Add a REAL16 field to a message using a name.

**C API:** TipcMsgAddNamedReal16

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_REAL16 **value*, T_INT4 *size*);`

**Remarks:** Add a field containing an array of REAL16 fields to a message using a name.

**C API:** TipcMsgAddNamedReal16Array

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_REAL16 **value*, T_INT4 *size*, T_IPC_MSG_FIELD **field_return*);`

**Remarks:** Add a field containing an array of REAL16 fields to a message using a name and a pointer to the field value, rather than a copy.

**C API:** TipcMsgAddNamedReal16ArrayPtr

**Syntax:** `T_BOOL AddNamed(T_STR name, T_STR value);`

**Remarks:** Add a STR field to a message using a name.

**C API:** TipcMsgAddNamedStr

**Syntax:** `T_BOOL AddNamed(T_STR name, T_STR *value, T_INT4 size);`

**Remarks:** Add a field containing an array of STR fields to a message using a name.

**C API:** TipcMsgAddNamedStrArray

**Syntax:**
```
T_BOOL AddNamed(T_STR name, T_STR *value, T_INT4 size,
                T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Add a field containing an array of STR fields to a message using a name and a pointer to the field value, rather than a copy.

**C API:** TipcMsgAddNamedStrArrayPtr

**Syntax:** `T_BOOL AddNamed(T_STR name, T_PTR ptr);`

**Remarks:** Add a STR field to a message using a name and a pointer to the field value, rather than a copy.

**C API:** TipcMsgAddNamedStrPtr

**Syntax:** `T_BOOL AddNamedUnknown(T_STR name, T_IPC_FT type);`

**Remarks:** Add a field with an unknown or non-existent value to a message using a name.

**C API:** TipcMsgAddNamedUnknown

**Syntax:** `T_BOOL AddNamedUtf8(T_STR name, T_STR value);`

**Remarks:** Add a UTF8 field to a message using a name.

**C API:** TipcMsgAddNamedUtf8

**Syntax:** `T_BOOL AddNamedUtf8(T_STR name, T_STR *value, T_INT4 size);`

**Remarks:** Add a field containing an array of UTF8 fields to a message using a name.

**C API:** TipcMsgAddNamedUtf8Array

**Syntax:** `T_BOOL AddNamedUtf8(T_STR *name*, T_STR *value, T_INT4 *size*, T_IPC_MSG_FIELD *field_return);`

**Remarks:** Add a field containing an array of UTF8 fields to a message using a name and a pointer to the field value, rather than a copy.

**C API:** TipcMsgAddNamedUtf8ArrayPtr

**Syntax:** `T_BOOL AddNamedUtf8(T_STR *name*, T_STR *value*, T_IPC_MSG_FIELD *field_return);`

**Remarks:** Add a UTF8 field to a message using a name and a pointer to the field value, rather than a copy.

**C API:** TipcMsgAddNamedUtf8Ptr

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_XML *value*);`

**Remarks:** Add the named XML field to a message.

**C API:** TipcMsgAddNamedXml

**Syntax:** `T_BOOL AddNamed(T_STR *name*, T_XML *value*, T_IPC_MSG_FIELD *field_return*);`

**Remarks:** Add the named XML field pointer to a message.

**C API:** TipcMsgAddNamedXmlPtr

### TipcMsg::Append

**Syntax:** `virtual T_BOOL Append(T_PTR *binary_data*, T_INT4 *binary_size*);`

**Remarks:** Append a BINARY field to a message.

**C API:** TipcMsgAppendBinary

**Syntax:** `virtual T_BOOL AppendBool(T_BOOL *bool_data*);`

**Remarks:** Append a BOOL field to a message.

**C API:** TipcMsgAppendBool

**Syntax:** `virtual T_BOOL AppendBool(T_BOOL *bool_array_data*, T_INT4 *array_size*);`

**Remarks:** Append a BOOL_ARRAY field to a message.

**C API:** TipcMsgAppendBoolArray

**Syntax:** `virtual T_BOOL Append(T_UCHAR byte_data, T_INT4 byte_size);`

**Remarks:** Append a BYTE field to a message.

**C API:** TipcMsgAppendByte

**Syntax:** `virtual T_BOOL Append(T_CHAR char_data);`

**Remarks:** Append a CHAR field to a message.

**C API:** TipcMsgAppendChar

**Syntax:** `virtual T_BOOL Append(T_INT2 int2_data);`

**Remarks:** Append an INT2 field to a message.

**C API:** TipcMsgAppendInt2

**Syntax:**
```
virtual T_BOOL Append(T_INT2 *int2_array_data,
                      T_INT4 int2_array_size);
```

**Remarks:** Append an INT2_ARRAY field to a message.

**C API:** TipcMsgAppendInt2Array

**Syntax:** `virtual T_BOOL Append(T_INT4 int4_data);`

**Remarks:** Append an INT4 field to a message.

**C API:** TipcMsgAppendInt4

**Syntax:**
```
virtual T_BOOL Append(T_INT4 *int4_array_data,
                      T_INT4 int4_array_size);
```

**Remarks:** Append an INT4_ARRAY field to a message.

**C API:** TipcMsgAppendInt4Array

**Syntax:** `virtual T_BOOL Append(T_INT8 int8_data);`

**Remarks:** Append an INT8 field to a message.

**C API:** TipcMsgAppendInt8

**Syntax:**
```
virtual T_BOOL Append(T_INT8 *int8_array_data,
                      T_INT4 int8_array_size);
```

**Remarks:** Append an INT8_ARRAY field to a message.

**C API:** TipcMsgAppendInt8Array

**Syntax:** `virtual T_BOOL Append(TipcMsg& msg_data);`

**Remarks:** Append the internal MSG field of a TipcMsg object to a message.

**C API:** TipcMsgAppendMsg

**Syntax:** `virtual T_BOOL Append(TipcMsg& *msg_array_data, T_INT4 msg_array_size);`

**Remarks:** Append a TipcMsg object array to message. The internal T_IPC_MSG of each object of the array is transformed in an array of MSG messages and inserted into a message.

**C API:** TipcMsgAppendMsgArray

**Syntax:** `virtual T_BOOL Append(T_REAL4 real4_data);`

**Remarks:** Append a REAL4 field to a message.

**C API:** TipcMsgAppendReal4

**Syntax:** `virtual T_BOOL Append(T_REAL4 *real4_array_data, T_INT4 real4_array_size);`

**Remarks:** Append a REAL4_ARRAY field to a message.

**C API:** TipcMsgAppendReal4Array

**Syntax:** `virtual T_BOOL Append(T_REAL8 real8_data);`

**Remarks:** Append a REAL8 field to message.

**C API:** TipcMsgAppendReal8

**Syntax:** `virtual T_BOOL Append(T_REAL8 *real8_array_data, T_INT4 real8_array_size);`

**Remarks:** Append a REAL8_ARRAY field to a message.

**C API:** TipcMsgAppendReal8Array

**Syntax:** `virtual T_BOOL Append(T_REAL16 real16_data);`

**Remarks:** Append a REAL16 field to a message.

**C API:** TipcMsgAppendReal16

**Syntax:** `virtual T_BOOL Append(T_REAL16 *real16_array_data, T_INT4 real16_array_size);`

**Remarks:** Append a REAL16_ARRAY field to a message.

**C API:** TipcMsgAppendReal16Array

**Syntax:** `virtual T_BOOL Append(T_STR str_data);`

**Remarks:** Append a STR field to a message.

**C API:** TipcMsgAppendStr

**Syntax:** `virtual T_BOOL Append(T_STR *str_array_data, T_INT4 str_array_size);`

**Remarks:** Append a STR_ARRAY field to a message.

**C API:** TipcMsgAppendStrArray

**Syntax:** `virtual T_BOOL Append(T_STR str_data, T_REAL8 real8_data);`

**Remarks:** Append a STR field and a REAL8 field to a message.

**C API:** TipcMsgAppendStrReal8

**Syntax:** `virtual T_BOOL AppendUtf8(T_STR str_data);`

**Remarks:** Append a UTF8 field to a message.

**C API:** TipcMsgAppendUtf8

**Syntax:** `virtual T_BOOL AppendUtf8(T_STR *str_array_data, T_INT4 array_size);`

**Remarks:** Append a UTF8_ARRAY field to a message.

**C API:** TipcMsgAppendUtf8Array

**Syntax:** `virtual T_BOOL Append(T_XML xml_data);`

**Remarks:** Append an XML field to a message.

**C API:** TipcMsgAppendXml

### TipcMsg::AppendPtr

**Syntax:**
```
virtual T_BOOL
AppendPtr(T_PTR binary_data, T_INT4 binary_data_size,
          T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Append a BINARY pointer field to a message.

**C API:** TipcMsgAppendBinaryPtr

**Syntax:**
```
virtual T_BOOL AppendBoolPtr(T_BOOL *bool_array_data,
                             T_INT4 guid_array_size,
                             T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Append a field containing an array of BOOL fields to a message using a pointer.

**C API:** TipcMsgAppendBoolArrayPtr

**Syntax:**
```
virtual T_BOOL AppendPtr(T_IPC_MSG msg_data,
                         T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Append a message pointer field to a message.

**C API:** TipcMsgAppendMsgPtr

**Syntax:**
```
virtual T_BOOL AppendPtr(T_INT2 *int2_array_data,
                         T_INT4 int2_array_size,
                         T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Append an INT2_ARRAY pointer field to a message.

**C API:** TipcMsgAppendInt2ArrayPtr

**Syntax:**
```
virtual T_BOOL AppendPtr(T_INT4 *int4_array_data,
                         T_INT4 int4_array_size,
                         T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Append an INT4_ARRAY pointer field to a message.

**C API:** TipcMsgAppendInt4ArrayPtr

**Syntax:**
```
virtual T_BOOL AppendPtr(T_INT8 *int8_array_data,
                         T_INT4 int8_array_size,
                         T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Append an INT8_ARRAY pointer field to a message.

**C API:** TipcMsgAppendInt8ArrayPtr

**Syntax:**
```
virtual T_BOOL AppendPtr(T_IPC_MSG *msg_array_data,
                         T_INT4 msg_array_size,
                         T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Append a MSG_ARRAY pointer field to a message.

**C API:** TipcMsgAppendMsgArrayPtr

**Syntax:**
```
virtual T_BOOL AppendPtr(T_REAL4 *real4_array_data,
                         T_INT4 real4_array_size,
                         T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Append a REAL4_ARRAY pointer field to a message.

**C API:** TipcMsgAppendReal4ArrayPtr

**Syntax:**
```
virtual T_BOOL AppendPtr(T_REAL8 *real8_array_data,
                         T_INT4 real8_array_size,
                         T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Append a REAL8_ARRAY pointer field to a message.

**C API:** TipcMsgAppendReal8ArrayPtr

**Syntax:**
```
virtual T_BOOL AppendPtr(T_REAL16 *real16_array_data,
                         T_INT4 real16_array_size,
                         T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Append a REAL16_ARRAY pointer field to a message.

**C API:** TipcMsgAppendReal16ArrayPtr

**Syntax:**
```
virtual T_BOOL AppendPtr(T_IPC_STR str_data,
                         T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Append a STR pointer field to a message.

**C API:** TipcMsgAppendStrPtr

**Syntax:**
```
virtual T_BOOL AppendPtr(T_STR *str_array_data,
                         T_INT4 str_array_size,
                         T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Append a STR_ARRAY pointer field to a message.

**C API:** TipcMsgAppendStrArrayPtr

**Syntax:**
```
virtual T_BOOL AppendUtf8Ptr(T_STR *str_array_data,
                             T_INT4 str_array_size,
                             T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Append a UTF8_ARRAY pointer field to a message.

**C API:** TipcMsgAppendUtf8ArrayPtr

**Syntax:** `virtual T_BOOL AppendUtf8Ptr(T_STR str_data, T_IPC_MSG_FIELD *field_return);`

**Remarks:** Append a UTF8 pointer field to a message.

**C API:** TipcMsgAppendUtf8Ptr

**Syntax:** `virtual T_BOOL AppendPtr(T_XML xml_data, T_IPC_MSG_FIELD *field_return);`

**Remarks:** Append an XML pointer field to a message.

**C API:** TipcMsgAppendXmlPtr

### TipcMsg::AppendUnknown

**Syntax:** `virtual T_BOOL AppendUnknown(T_IPC_FT field_type);`

**Remarks:** Appends a field of the given type with an unknown value.

**C API:** TipcMsgAppendUnknown

### TipcMsg::Clone

**Syntax:** `T_BOOL Clone(TipcMsg& msg);`

**Remarks:** Make a clone of a message and assign it to the internal T_IPC_MSG field of *msg*. See the detailed description of TipcMsgClone in the *TIBCO SmartSockets Application Programming Interface* for a detailed description of message cloning. Any message that the object was managing at the time of the call is passed to TipcMsgDestroy().

**C API:** TipcMsgClone

### TipcMsg::CorrelationId

**Syntax:** `T_STR CorrelationId();`

**Remarks:** Get the correlation identifier property of a message.

**C API:** TipcMsgGetCorrelationId

**Syntax:** `T_BOOL CorrelationId(T_STR correlation_id);`

**Remarks:** Set the correlation identifier property of a message.

**C API:** TipcMsgSetCorrelationId

### TipcMsg::Create

**Syntax:** `T_BOOL Create(TipcMt& mt);`

**Remarks:** Create a new message. Any message that the object was managing at the time of the call is passed to TipcMsgDestroy().

**C API:** TipcMsgCreate

**Syntax:** `T_BOOL Create(T_INT4 mt_num);`

**Remarks:** Create a new message using *mt_num*. *mt_num* is the message type number. Any message that the object was managing at the time of the call is passed to TipcMsgDestroy().

**C API:** TipcMsgCreate

**Syntax:** `T_BOOL Create(T_STR mt_name);`

**Remarks:** Create a new message using *mt_name*. *mt_name* is the message type name. Any message that the object was managing at the time of the call is passed to TipcMsgDestroy().

**C API:** TipcMsgCreate

### TipcMsg::Current

**Syntax:** `T_INT4 Current();`

**Remarks:** Get the current field of a message.

**C API:** TipcMsgGetCurrent

**Syntax:** `T_BOOL Current(T_INT4 field_num);`

**Remarks:** Set the current field of a message.

**C API:** TipcMsgSetCurrent

### TipcMsg::CurrentFieldKnown

**Syntax:** `T_BOOL CurrentFieldKnown();`

**Remarks:** Returns `TRUE` if the value of the current field is known.

**C API:** TipcMsgCurrentFieldKnown

### TipcMsg::DeleteCurrent

**Syntax:** `T_BOOL DeleteField();`

**Remarks:** Delete the current field in a message, whether it is named or unnamed.

**C API:** TipcMsgDeleteCurrent

### TipcMsg::DeleteField

**Syntax:** `T_BOOL DeleteField(T_INT4 field_num);`

**Remarks:** Delete the specified field in a message.

**C API:** TipcMsgDeleteField

### TipcMsg::DeleteNamedField

**Syntax:** `T_BOOL DeleteNamedField(T_STR name);`

**Remarks:** Delete the field in a message with the specified name.

**C API:** TipcMsgDeleteNamedField

### TipcMsg::DeliveryMode

**Syntax:** `T_IPC_DELIVERY_MODE DeliveryMode();`

**Remarks:** Get the delivery mode of a message.

**C API:** TipcMsgGetDeliveryMode

**Syntax:** `T_BOOL DeliveryMode(T_IPC_DELIVERY_MODE delivery_mode);`

**Remarks:** Set the delivery mode of a message.

**C API:** TipcMsgSetDeliveryMode

### TipcMsg::DeliveryTimeout

**Syntax:** `T_REAL8 DeliveryTimeout();`

**Remarks:** Get the delivery timeout of a message.

**C API:** TipcMsgGetDeliveryTimeout

**Syntax:** `T_BOOL DeliveryTimeout(T_REAL8 delivery_timeout);`

**Remarks:** Set the delivery timeout of a message.

**C API:** TipcMsgSetDeliveryTimeout

### TipcMsg::Dest

**Syntax:** `T_STR Dest();`

**Remarks:** Get the destination property of a message.

**C API:** TipcMsgGetDest

**Syntax:** `T_BOOL Dest(T_STR dest);`

**Remarks:** Set the destination property of a message.

**C API:** TipcMsgSetDest

### TipcMsg::Destroy

**Syntax:** `T_BOOL Destroy();`

**Remarks:** Destroy a message. This call leaves the object vacant after calling TipcMsgDestroy().

**C API:** TipcMsgDestroy

### TipcMsg::Expiration

**Syntax:** `T_REAL8 Expiration();`

**Remarks:** Get the time to live property of a message.

**C API:** TipcMsgGetExpiration

**Syntax:** `T_BOOL Expiration(T_REAL8 expiration);`

**Remarks:** Set the time to live property of a message.

**C API:** TipcMsgSetExpiration

### TipcMsg::GetNamed

**Syntax:** `T_BOOL GetNamed(T_STR name, T_PTR *value_return; T_INT4 *size_return);`

**Remarks:** Get a field containing a BINARY value from a message, using a name.

**C API:** TipcMsgGetNamedBinary

**Syntax:** `T_BOOL GetNamedBool(T_STR name, T_BOOL *value_return);`

**Remarks:** Get a field with a BOOL value from a message, using a name.

**C API:** TipcMsgGetNamedBool

**Syntax:** `T_BOOL GetNamedBool(T_STR name, T_BOOL **value_return, T_INT4 *size_return);`

**Remarks:** Get a field containing an array of BOOL values from a message, using a name.

**C API:** TipcMsgGetNamedBoolArray

**Syntax:** `T_BOOL GetNamed(T_STR name, T_UCHAR *value_return);`

**Remarks:** Get a field with a BYTE value from a message, using a name.

**C API:** TipcMsgGetNamedByte

**Syntax:** `T_BOOL GetNamed( T_STR name, T_CHAR *value_return);`

**Remarks:** Get a field with a character value from a message, using a name.

**C API:** TipcMsgGetNamedChar

**Syntax:** `T_BOOL GetNamed(T_STR name, T_INT2 *value_return);`

**Remarks:** Get a field with an INT2 value from a message, using a name.

**C API:** TipcMsgGetNamedInt2

**Syntax:** `T_BOOL GetNamed(T_STR name, T_INT2 **value_return, T_INT4 *size_return);`

**Remarks:** Get a field containing an array of INT2 values from a message, using a name.

**C API:** TipcMsgGetNamedInt2Array

**Syntax:** `T_BOOL GetNamed(T_STR *name*, T_INT4 **value_return*);`

**Remarks:** Get a field with an INT4 value from a message, using a name.

**C API:** TipcMsgGetNamedInt4

**Syntax:** `T_BOOL GetNamed(T_STR *name*, T_INT4 ***value_return*, T_INT4 **size_return*);`

**Remarks:** Get a field containing an array of INT4 values from a message, using a name.

**C API:** TipcMsgGetNamedInt4Array

**Syntax:** `T_BOOL GetNamed(T_STR *name*, T_INT8 **value_return*);`

**Remarks:** Get a field with an INT8 value from a message, using a name.

**C API:** TipcMsgGetNamedInt8

**Syntax:** `T_BOOL GetNamed(T_STR *name*, T_INT8 ***value_return*, T_INT4 **size_return*);`

**Remarks:** Get a field containing an array of INT8 values from a message, using a name.

**C API:** TipcMsgGetNamedInt8Array

**Syntax:** `T_BOOL GetNamed(T_STR *name*, T_IPC_MSG **value_return*);`

**Remarks:** Get a field with a MSG value from a message, using a name.

**C API:** TipcMsgGetNamedMsg

**Syntax:** `T_BOOL GetNamed(T_STR *name*, T_IPC_MSG ***value_return*, T_INT4 **size_return*);`

**Remarks:** Get a field containing an array of MSG values from a message, using a name.

**C API:** TipcMsgGetNamedMsgArray

**Syntax:** `T_BOOL GetNamed(T_STR *name*, T_REAL4 **value_return*);`

**Remarks:** Get a field with a REAL4 value from a message, using a name.

**C API:** TipcMsgGetNamedReal4

**Syntax:** `T_BOOL GetNamed(T_STR *name*, T_REAL4 **value_return*, T_INT4 **size_return*);`

**Remarks:** Get a field containing an array of REAL4 values from a message, using a name.

**C API:** TipcMsgGetNamedReal4Array

**Syntax:** `T_BOOL GetNamed(T_STR *name*, T_REAL8 **value_return*);`

**Remarks:** Get a field with a REAL8 value from a message, using a name.

**C API:** TipcMsgGetNamedReal8

**Syntax:** `T_BOOL GetNamed(T_STR *name*, T_REAL8 **value_return*, T_INT4 **size_return*);`

**Remarks:** Get a field containing an array of REAL8 values from a message, using a name.

**C API:** TipcMsgGetNamedReal8Array

**Syntax:** `T_BOOL GetNamed(T_STR *name*, T_REAL16 **value_return*);`

**Remarks:** Get a field with a REAL16 value from a message, using a name.

**C API:** TipcMsgGetNamedReal16

**Syntax:** `T_BOOL GetNamed(T_STR *name*, T_REAL16 **value_return*, T_INT4 **size_return*);`

**Remarks:** Get a field containing an array of REAL16 values from a message, using a name.

**C API:** TipcMsgGetNamedReal16Array

**Syntax:** `T_BOOL GetNamed(T_STR *name*, T_STR **value_return*);`

**Remarks:** Get a field with a STR value from a message, using a name.

**C API:** TipcMsgGetNamedStr

**Syntax:** `T_BOOL GetNamed(T_STR *name*, T_STR **value_return*, T_INT4 **size_return*);`

**Remarks:** Get a field containing an array of STR values from a message, using a name.

**C API:** TipcMsgGetNamedStrArray

**Syntax:** `T_BOOL GetNamedUnknown(T_STR name);`

**Remarks:** Get a field with an unknown or non-existent value from a message, using a name.

**C API:** TipcMsgGetNamedUnknown

**Syntax:** `T_BOOL GetNamedUtf8(T_STR name, T_STR *value_return);`

**Remarks:** Get a field with a UTF8 value from a message, using a name.

**C API:** TipcMsgGetNamedUtf8

**Syntax:** `T_BOOL GetNamedUtf8(T_STR name, T_STR **value_return, T_INT4 *size_return);`

**Remarks:** Get a field containing an array of UTF8 values from a message, using a name.

**C API:** TipcMsgGetNamedUtf8Array

**Syntax:** `T_BOOL GetNamed(T_STR name, T_XML *xml_return);`

**Remarks:** Get a field containing an XML value from a message, using a name.

**C API:** TipcMsgGetNamedXml

### TipcMsg::HeaderStrEncode

**Syntax:** `T_BOOL HeaderStrEncode();`

**Remarks:** Get the header string encoding mode of a message.

**C API:** TipcMsgGetHeaderStrEncode

**Syntax:** `T_BOOL HeaderStrEncode(T_BOOL header_str_encode);`

**Remarks:** Set the header string encoding mode of a message.

**C API:** TipcMsgSetHeaderStrEncode

### TipcMsg::IncrRefCount

**Syntax:** `T_BOOL IncrRefCount();`

**Remarks:** Increment the reference count of a message.

**C API:** TipcMsgIncrRefCount

### TipcMsg::LbMode

**Syntax:** `T_IPC_LB_MODE LbMode();`

**Remarks:** Get the load balancing mode of a message.

**C API:** TipcMsgGetLbMode

**Syntax:** `T_BOOL LbMode(T_IPC_LB_MODE` *lb_mode*`);`

**Remarks:** Set the load balancing mode of a message.

**C API:** TipcMsgSetLbMode

### TipcMsg::Message

**Syntax:** `T_IPC_MSG Message();`

**Remarks:** Return the internal T_IPC_MSG of the TipcMsg object.

### TipcMsg::MessageId

**Syntax:** `T_STR MessageId();`

**Remarks:** Get the message identifier property of a message.

**C API:** TipcMsgGetMessageId

**Syntax:** `T_BOOL GenerateMessageId();`

**Remarks:** Generate the message identifier property for the message.

**C API:** TipcMsgGenerateMessageId

### TipcMsg::Next

**Syntax:** `virtual T_BOOL Next(T_PTR *`*binary_data*`, T_INT4 *`*binary_size*`);`

**Remarks:** Get a BINARY field from a message.

**C API:** TipcMsgNextBinary

**Syntax:** `virtual T_BOOL NextBool(T_BOOL *`*bool_return*`);`

**Remarks:** Get a BOOL field from a message.

**C API:** TipcMsgNextBool

**Syntax:** `virtual T_BOOL NextBool(T_BOOL **bool_array_data, T_INT4 *array_size);`

**Remarks:** Get a BOOL_ARRAY field from a message.

**C API:** TipcMsgNextBoolArray

**Syntax:** `virtual T_BOOL Next(T_UCHAR byte_return);`

**Remarks:** Get a BYTE field from a message.

**C API:** TipcMsgNextByte

**Syntax:** `virtual T_BOOL Next(T_CHAR *char_data);`

**Remarks:** Get a CHAR field from a message.

**C API:** TipcMsgNextChar

**Syntax:** `virtual T_BOOL Next(T_INT2 *int2_data);`

**Remarks:** Get an INT2 field from a message.

**C API:** TipcMsgNextInt2

**Syntax:** `virtual T_BOOL Next(T_INT2 **int2_array_data, T_INT4 *int2_array_size);`

**Remarks:** Get an INT2_ARRAY field from a message.

**C API:** TipcMsgNextInt2Array

**Syntax:** `virtual T_BOOL Next(T_INT4 *int4_data);`

**Remarks:** Get an INT4 field from a message.

**C API:** TipcMsgNextInt4

**Syntax:** `virtual T_BOOL Next(T_INT4 **int4_array_data, T_INT4 *int4_array_size);`

**Remarks:** Get an INT4_ARRAY field from a message.

**C API:** TipcMsgNextInt4Array

**Syntax:** `virtual T_BOOL Next(T_INT8 *int8_data);`

**Remarks:** Get an INT8 field from a message.

**C API:** TipcMsgNextInt8

**Syntax:** `virtual T_BOOL Next(TipcMsg **msg_array_data, T_INT4 *msg_array_size);`

**Remarks:** Get a MSG_ARRAY field from a message and assign each element of the array to an element of a TipcMsg object array.

**C API:** TipcMsgNextMsgArray

**Syntax:** `virtual T_BOOL Next(TipcMsg *msg_data);`

**Remarks:** Get a message field from a message and assign it to the internal field of a TipcMsg object.

**C API:** TipcMsgNextMsg

**Syntax:** `virtual T_BOOL Next(T_INT8 **int8_data, T_INT4 *int8_array_size);`

**Remarks:** Get an INT8_ARRAY field from a message.

**C API:** TipMsgNextInt8Array

**Syntax:** `virtual T_BOOL Next(T_REAL4 *real4_data);`

**Remarks:** Get a REAL4 field from a message.

**C API:** TipcMsgNextReal4

**Syntax:** `virtual T_BOOL Next(T_REAL4 **real4_array_data, T_INT4 *real4_array_size);`

**Remarks:** Get a REAL4_ARRAY field from a message.

**C API:** TipcMsgNextReal4Array

**Syntax:** `virtual T_BOOL Next(T_REAL8 *real8_data);`

**Remarks:** Get a REAL8 field from a message.

**C API:** TipcMsgNextReal8

**Syntax:** `virtual T_BOOL Next(T_REAL8 **real8_array_data, T_INT4 *real8_array_size);`

**Remarks:** Get a REAL8_ARRAY field from a message.

**C API:** TipcMsgNextReal8Array

**Syntax:** `virtual T_BOOL Next(T_REAL16 *real16_data);`

**Remarks:** Get a REAL16 field from a message

**C API:** TipcMsgNextReal16

**Syntax:**
```
virtual T_BOOL Next(T_REAL16 **real16_array_data,
                    T_INT4 *real16_array_size);
```

**Remarks:** Get a REAL16_ARRAY field from a message

**C API:** TipcMsgNextReal16Array

**Syntax:** `virtual T_BOOL Next(T_STR *str_data);`

**Remarks:** Get a STR field from a message.

**C API:** TipcMsgNextStr

**Syntax:**
```
virtual T_BOOL Next(T_STR **str_array_data,
                    T_INT4 *str_array_size);
```

**Remarks:** Get a STR_ARRAY field from a message.

**C API:** TipcMsgNextStrArray

**Syntax:** `virtual T_BOOL Next(T_STR *str_data, T_REAL8 *real8_data);`

**Remarks:** Get a STR field and a REAL8 field from a message.

**C API:** TipcMsgNextStrReal8

**Syntax:** `virtual T_BOOL NextUtf8(T_STR *str_data);`

**Remarks:** Get a UTF8 field from a message.

**C API:** TipcMsgNextUtf8

**Syntax:**
```
virtual T_BOOL NextUtf8(T_STR **str_array_data,
                        T_INT4 *array_size);
```

**Remarks:** Get a UTF_ARRAY field from a message.

**C API:** TipcMsgNextUtf8

**Syntax:** `virtual T_BOOL Next(T_XML *xml_data)`

**Remarks:** Get an XML string from a message.

**C API:** TipcMsgNextXml

### TipcMsg::NextUnknown

**Syntax:** `T_BOOL NextUnknown();`

**Remarks:** Get an unknown field from a message.

**C API:** TipcMsgNextUnknown

### TipcMsg::NumFields

**Syntax:** `T_INT4 NumFields();`

**Remarks:** Get the number of fields in a message.

**C API:** TipcMsgGetNumFields

**Syntax:** `T_BOOL NumFields(T_INT4 num_fields);`

**Remarks:** Set the number of fields in a message.

**C API:** TipcMsgSetNumFields

### TipcMsg::PacketSize

**Syntax:** `T_UINT4 PacketSize();`

**Remarks:** Get the packet size of a message.

**C API:** TipcMsgGetPacketSize

### TipcMsg::Print

**Syntax:** `virtual T_BOOL Print(T_OUT_FUNC func);`

**Remarks:** Print all information in a message.

**C API:** TipcMsgPrint

### TipcMsg::PrintError

**Syntax:** `virtual T_BOOL PrintError();`

**Remarks:** Report an error about an unexpected message.

**C API:** TipcMsgPrintError

### TipcMsg::Priority

**Syntax:** `T_INT2 Priority();`

**Remarks:** Get the priority of a message.

**C API:** TipcMsgGetPriority

**Syntax:** `T_BOOL Priority(T_INT2 priority);`

**Remarks:** Set the priority of a message.

**C API:** TipcMsgSetPriority

### TipcMsg::ReadOnly

**Syntax:** `T_BOOL ReadOnly();`

**Remarks:** Get the read-only status of a message.

**C API:** TipcMsgGetReadOnly

### TipcMsg::RefCount

**Syntax:** `T_INT4 RefCount();`

**Remarks:** Get the reference count of a message.

**C API:** TipcMsgGetRefCount

### TipcMsg::ReplyTo

**Syntax:** `T_STR ReplyTo();`

**Remarks:** Get the reply to destination property of a message.

**C API:** TipcMsgGetReplyTo

**Syntax:** `T_BOOL ReplyTo(T_STR subj);`

**Remarks:** Set the reply to destination property of a message.

**C API:** TipcMsgSetReplyTo

### TipcMsg::ResetCheck

**Syntax:** `void ResetCheck();`

**Remarks:** Reset the internal problem flag used by insertion and extraction operators to FALSE. See also the description of the Check manipulator in the Manipulators section on page 151.

### TipcMsg::Sender

**Syntax:** `T_STR Sender();`

**Remarks:** Get the sender property of a message.

**C API:** TipcMsgGetSender

**Syntax:** `T_BOOL Sender(T_STR` *sender*`);`

**Remarks:** Set the sender property of a message.

**C API:** TipcMsgSetSender

### TipcMsg::SeqNum

**Syntax:** `T_INT4 SeqNum();`

**Remarks:** Get the sequence number of a message.

**C API:** TipcMsgGetSeqNum

### TipcMsg::Timestamp

**Syntax:** `T_REAL8 SenderTimestamp();`

**Remarks:** Get the sender timestamp property of a message.

**C API:** TipcMsgGetSenderTimestamp

**Syntax:** `T_BOOL SenderTimestamp(T_REAL8` *timestamp*`);`

**Remarks:** Set the sender timestamp property of a message.

**C API:** TipcMsgSetSenderTimestamp

**Syntax:** `T_REAL8 ArrivalTimestamp();`

**Remarks:** Get the arrival timestamp property of a message.

**C API:** TipcMsgGetArrivalTimestamp

**Syntax:** `T_BOOL ArrivalTimestamp(T_REAL8 timestamp);`

**Remarks:** Set the arrival timestamp property of a message.

**C API:** TipcMsgSetArrivalTimestamp

### TipcMsg::Traverse

**Syntax:** `T_PTR Traverse(T_IPC_MSG_TRAV_FUNC func, T_PTR arg);`

**Remarks:** Traverse all fields in a message.

**C API:** TipcMsgTraverse

### TipcMsg::Type

**Syntax:** `TipcMt& Type();`

**Remarks:** Get the type of a message.

**C API:** TipcMsgGetType

**Syntax:** `T_BOOL Type(TipcMt& mt);`

**Remarks:** Set the type of a message.

**C API:** TipcMsgSetType

### TipcMsg::UpdateNamed

**Syntax:** `T_BOOL UpdateNamed(T_STR name, T_PTR value, T_INT4 size);`

**Remarks:** Update a named BINARY field in a message.

**C API:** TipcMsgUpdateNamedBinary

**Syntax:**
```
T_BOOL UpdateNamed(T_STR name, T_PTR value, T_INT4 size,
                   T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Update a named BINARY pointer field in a message.

**C API:** TipcMsgUpdateNamedBinaryPtr

**Syntax:** `T_BOOL UpdateNamedBool(T_STR name, T_BOOL value);`

**Remarks:** Update a named BOOL field in a message.

**C API:** TipcMsgUpdateNamedBool

**Syntax:**
```
T_BOOL UpdateNamedBool(T_STR name, T_BOOL *value,
                       T_INT4 size);
```

**Remarks:** Update a named BOOL_ARRAY field in a message.

**C API:** TipcMsgUpdateNamedBoolArray

**Syntax:**
```
T_BOOL UpdateNamedBool(T_STR name, T_BOOL *value,
                       T_INT4 size,
                       T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Update a named BOOL_ARRAY pointer field in a message.

**C API:** TipcMsgUpdateNamedBoolArrayPtr

**Syntax:** `T_BOOL UpdateNamed(T_STR name, T_UCHAR value);`

**Remarks:** Update a named BYTE field in a message.

**C API:** TipcMsgUpdateNamedByte

**Syntax:** `T_BOOL UpdateNamed(T_STR name, T_CHAR value);`

**Remarks:** Update a named CHAR data field in a message.

**C API:** TipcMsgUpdateNamedChar

**Syntax:** `T_BOOL UpdateNamed(T_STR name, T_INT2 value);`

**Remarks:** Update a named INT2 data field in a message.

**C API:** TipcMsgUpdateNamedInt2

**Syntax:** `T_BOOL UpdateNamed(T_STR name, T_INT2 *value, T_INT4 size);`

**Remarks:** Update a named INT2_ARRAY field in a message.

**C API:** TipcMsgUpdateNamedInt2Array

**Syntax:**
```
T_BOOL UpdateNamed(T_STR name, T_INT2 *value, T_INT4 size,
                   T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Update a named INT2_ARRAY pointer field in a message.

**C API:** TipcMsgUpdateNamedInt2ArrayPtr

**Syntax:** `T_BOOL UpdateNamed(T_STR *name*, T_INT4 *value*);`

**Remarks:** Update a named INT4 data field in a message.

**C API:** TipcMsgUpdateNamedInt4

**Syntax:** `T_BOOL UpdateNamed(T_STR *name*, T_INT4 **value*, T_INT4 *size*);`

**Remarks:** Update a named INT4_ARRAY field in a message.

**C API:** TipcMsgUpdateNamedInt4Array

**Syntax:** `T_BOOL UpdateNamed(T_STR *name*, T_INT4 **value*, T_INT4 *size*, T_IPC_MSG_FIELD **field_return*);`

**Remarks:** Update a named INT4_ARRAY pointer field in a message.

**C API:** TipcMsgUpdateNamedInt4ArrayPtr

**Syntax:** `T_BOOL UpdateNamed(T_STR *name*, T_INT8 *value*);`

**Remarks:** Update a named INT8 field in a message.

**C API:** TipcMsgUpdateNamedInt8

**Syntax:** `T_BOOL UpdateNamed(T_STR *name*, T_INT8 **value*, T_INT4 *size*);`

**Remarks:** Update a named INT8_ARRAY field in a message.

**C API:** TipcMsgUpdateNamedInt8Array

**Syntax:** `T_BOOL UpdateNamed(T_STR *name*, T_INT8 **value*, T_INT4 *size*, T_IPC_MSG_FIELD **field_return*);`

**Remarks:** Update a named INT8_ARRAY pointer field in a message.

**C API:** TipcMsgUpdateNamedInt8ArrayPtr

**Syntax:** `T_BOOL UpdateNamed(T_STR *name*, T_IPC_MSG *value*);`

**Remarks:** Update a named MSG field in a message.

**C API:** TipcMsgUpdateNamedMsg

**Syntax:** `T_BOOL UpdateNamed(T_STR *name*, T_IPC_MSG **value*, T_INT4 *size*);`

**Remarks:** Update a named MSG array data field in a message.

**C API:** TipcMsgUpdateNamedMsgArray

**Syntax:** `T_BOOL pdateNamed(T_STR name, T_IPC_MSG value, T_IPC_MSG_FIELD *field_return);`

**Remarks:** Update a named MSG pointer field in a message.

**C API:** TipcMsgUpdateNamedMsgPtr

**Syntax:** `T_BOOL UpdateNamed(T_STR name, T_IPC_MSG *value, T_INT4 size, T_IPC_MSG_FIELD *field_return);`

**Remarks:** Update a named MSG_ARRAY pointer field in a message.

**C API:** TipcMsgUpdateNamedMsgArrayPtr

**Syntax:** `T_BOOL UpdateNamed(T_STR name, T_REAL4 value);`

**Remarks:** Update a named REAL4 data field in a message.

**C API:** TipcMsgUpdateNamedReal4

**Syntax:** `T_BOOL UpdateNamed(T_STR name, T_REAL4 *value, T_INT4 size);`

**Remarks:** Update a named REAL4_ARRAY field in a message.

**C API:** TipcMsgUpdateNamedReal4Array

**Syntax:** `T_BOOL UpdateNamed(T_STR name, T_REAL4 *value, T_INT4 size, T_IPC_MSG_FIELD *field_return);`

**Remarks:** Update a named REAL4_ARRAY pointer field in a message.

**C API:** TipcMsgUpdateNamedReal4ArrayPtr

**Syntax:** `T_BOOL UpdateNamed(T_STR name, T_REAL8 value);`

**Remarks:** Update a named REAL8 data field in a message.

**C API:** TipcMsgUpdateNamedReal8

**Syntax:** `T_BOOL UpdateNamed(T_STR name, T_REAL8 *value, T_INT4 size);`

**Remarks:** Update a named REAL8_ARRAY field in a message.

**C API:** TipcMsgUpdateNamedReal8Array

**Syntax:** `T_BOOL UpdateNamed(T_STR name, T_REAL8 *value, T_INT4 size, T_IPC_MSG_FIELD *field_return);`

**Remarks:** Update a named REAL8_ARRAY pointer field in a message.

**C API:** TipcMsgUpdateNamedReal8ArrayPtr

**Syntax:** `T_BOOL UpdateNamed(T_STR name, T_REAL16 value);`

**Remarks:** Update a named REAL16 data field in a message.

**C API:** TipcMsgUpdateNamedReal16

**Syntax:** `T_BOOL UpdateNamed(T_STR name, T_REAL16 *value, T_INT4 size);`

**Remarks:** Update a named REAL16_ARRAY field in a message.

**C API:** TipcMsgUpdateNamedReal16Array

**Syntax:**
```
T_BOOL UpdateNamed(T_STR name, T_REAL16 *value, T_INT4 size,
                   T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Update a named REAL16_ARRAY pointer field in a message.

**C API:** TipcMsgUpdateNamedReal16ArrayPtr

**Syntax:** `T_BOOL UpdateNamed(T_STR name, T_STR value);`

**Remarks:** Update a named STR field in a message.

**C API:** TipcMsgUpdateNamedStr

**Syntax:**
```
T_BOOL UpdateNamed(T_STR name, T_STR value,
                   T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Update a named STR pointer field in a message.

**C API:** TipcMsgUpdateNamedStrPtr

**Syntax:** `T_BOOL UpdateNamed(T_STR name, T_STR *value, T_INT4 size);`

**Remarks:** Update a named STR_ARRAY field in a message.

**C API:** TipcMsgUpdateNamedStrArray

**Syntax:**
```
T_BOOL UpdateNamed(T_STR name, T_STR *value, T_INT4 size,
                   T_IPC_MSG_FIELD *field_return);
```

**Remarks:** Update a named STR_ARRAY pointer field in a message.

**C API:** TipcMsgUpdateNamedStrArrayPtr

**Syntax:** `T_BOOL UpdateNamedUtf8(T_STR name, T_STR value);`

**Remarks:** Update a named UTF8 data field in a message.

**C API:** TipcMsgUpdateNamedUtf8

**Syntax:** `T_BOOL UpdateNamedUtf8(T_STR *name*, T_STR *value*, T_INT4 *size*);`

**Remarks:** Update a named UTF8_ARRAY field in a message.

**C API:** TipcMsgUpdateNamedUtf8Array

**Syntax:** `T_BOOL UpdateNamedUtf8(T_STR *name*, T_STR *value*, T_INT4 *size*, T_IPC_MSG_FIELD *field_return*);`

**Remarks:** Update a named UTF8_ARRAY pointer field in a message.

**C API:** TipcMsgUpdateNamedUtf8ArrayPtr

**Syntax:** `T_BOOL UpdateNamedUtf8(T_STR *name*, T_STR *value*, T_IPC_MSG_FIELD *field_return*);`

**Remarks:** Update a named UTF8 pointer field in a message.

**C API:** TipcMsgUpdateNamedUtf8Ptr

**Syntax:** `T_BOOL UpdateNamed(T_STR *name*, T_XML *value*);`

**Remarks:** Update a named XML data field in a message.

**C API:** TipcMsgUpdateNamedXml

**Syntax:** `T_BOOL UpdateNamed(T_STR *name*, T_XML *value*, T_IPC_MSG_FIELD *field_return*);`

**Remarks:** Update a named XML pointer field in a message.

**C API:** TipcMsgUpdateNamedXmlPtr

### TipcMsg::UserProp

**Syntax:** `T_INT4 UserProp();`

**Remarks:** Get user-defined property from a message.

**C API:** TipcMsgGetUserProp

**Syntax:** `T_BOOL UserProp(T_INT4 *user_prop*);`

**Remarks:** Set user-defined property of a message.

**C API:** TipcMsgSetUserProp

### TipcMsg::Vacant

**Syntax:** `T_BOOL Vacant();`

**Remarks:** Return `TRUE` if the object is vacant, `FALSE` otherwise.

## Conversion Operators

### TipcMsg::T_IPC_MSG

**Syntax:** `operator T_IPC_MSG() const;`

**Remarks:** Convert a TipcMsg object to a C API T_IPC_MSG by returning the internal T_IPC_MSG of the TipcMsg object.

## Operators

### TipcMsg::operator=

**Syntax:** `TipcMsg& operator=(const T_IPC_MSG` *msg*`);`

**Remarks:** Assign a T_IPC_MSG message to a TipcMsg object. *msg* is a SmartSockets C API T_IPC_MSG message assigned to the internal T_IPC_MSG message of a TipcMsg object. The reference count for that particular T_IPC_MSG message is incremented by calling TipcMsgIncrRefCount(). Any message that the object was managing at the time of the call is passed to TipcMsgDestroy().

**Syntax:** `TipcMsg& operator=(const TipcMsg&` *msg_obj*`);`

**Remarks:** Assign the T_IPC_MSG message contained in *msg_obj* to another TipcMsg object. The T_IPC_MSG pointer of the assigned object is assigned the value of the internal T_IPC_MSG pointer of *msg_obj*, and the reference count for that particular T_IPC_MSG message is incremented by calling TipcMsgIncrRefCount(). Any message that the object was managing at the time of the call is passed to TipcMsgDestroy().

### TipcMsg::operator<<

**Syntax:** `virtual TipcMsg& operator<<(T_PTR arg);`
**Remarks:** Insert a BINARY data field into a message.
**C API:** TipcMsgAppendBinary

**Syntax:** `virtual TipcMsg& operator<<(T_CHAR arg);`
**Remarks:** Insert a CHAR field into a message.
**C API:** TipcMsgAppendChar

**Syntax:** `virtual TipcMsg& operator<<(T_INT2 arg);`
**Remarks:** Insert an INT2 field into a message.
**C API:** TipcMsgAppendInt2

**Syntax:** `virtual TipcMsg& operator<<(T_INT2 *arg);`
**Remarks:** Insert an INT2_ARRAY field into a message.
**C API:** TipcMsgAppendInt2Array

**Syntax:** `virtual TipcMsg& operator<<(T_INT4 arg);`
**Remarks:** Insert an INT4 field into a message.
**C API:** TipcMsgAppendInt4

**Syntax:** `virtual TipcMsg& operator<<(T_INT4 *arg);`
**Remarks:** Insert an INT4_ARRAY field into a message.
**C API:** TipcMsgAppendInt4Array

**Syntax:** `virtual TipcMsg& operator<<(T_INT8 arg);`
**Remarks:** Insert an INT8 field into a message.
**C API:** TipcMsgAppendInt8

**Syntax:** `virtual TipcMsg& operator<<(T_INT8 *arg);`
**Remarks:** Insert an INT8_ARRAY field into a message.
**C API:** TipcMsgAppendInt8Array

**Syntax:** `virtual TipcMsg& operator<<(TipcMsg& arg);`

**Remarks:** Insert a message from another TipcMsg object into a message. The T_IPC_MSG contained in the TipcMsg object argument is inserted into the message being operated on by using TipcMsgAppendMsg.

**C API:** TipcMsgAppendMsg

**Syntax:** `virtual TipcMsg& operator<<(TipcMsg *arg);`

**Remarks:** Insert a TipcMsg array into a message. The internal T_IPC_MSG of each object of the array is transformed into an array of MSG messages and inserted into a message.

**C API:** TipcMsgAppendMsgArray

**Syntax:** `virtual TipcMsg& operator<<(T_REAL4 arg);`

**Remarks:** Insert a REAL4 field into a message.

**C API:** TipcMsgAppendReal4

**Syntax:** `virtual TipcMsg& operator<<(T_REAL4 *arg);`

**Remarks** Insert a REAL4_ARRAY field into a message.

**C API:** TipcMsgAppendReal4Array

**Syntax:** `virtual TipcMsg& operator<<(T_REAL8 arg);`

**Remarks:** Insert a REAL8 field into a message.

**C API:** TipcMsgAppendReal8

**Syntax:** `virtual TipcMsg& operator<<(T_REAL8 *arg);`

**Remarks:** Insert a REAL8_ARRAY field into a message.

**C API:** TipcMsgAppendReal8Array

**Syntax:** `virtual TipcMsg& operator<<(T_REAL16 arg);`

**Remarks:** Insert a REAL16 field into a message.

**C API:** TipcMsgAppendReal16

**Syntax:** `virtual TipcMsg& operator<<(T_REAL16 *arg);`
**Remarks:** Insert a REAL16_ARRAY field into a message.
**C API:** TipcMsgAppendReal16Array

**Syntax:** `virtual TipcMsg& operator<<(T_STR arg);`
**Remarks:** Insert a STR field into a message.
**C API:** TipcMsgAppendStr

**Syntax:** `virtual TipcMsg& operator<<(T_STR *arg);`
**Remarks:** Insert a STR_ARRAY field into a message.
**C API:** TipcMsgAppendStrArray

**Syntax:** `virtual TipcMsg& operator<<(T_XML *arg);`
**Remarks:** Insert an XML field into a message.
**C API:** TipcMsgAppendXml

### TipcMsg::operator>>

**Syntax:** `virtual TipcMsg& operator>>(T_PTR& arg);`
**Remarks:** Extract a BINARY data field from a message.
**C API:** TipcMsgNextBinary

**Syntax:** `virtual TipcMsg& operator>>(T_CHAR& arg);`
**Remarks:** Extract a CHAR field from a message.
**C API:** TipcMsgNextChar

**Syntax:** `virtual TipcMsg& operator>>(T_INT2& arg);`
**Remarks:** Extract an INT2 field from a message.
**C API:** TipcMsgNextInt2

**Syntax:** `virtual TipcMsg& operator>>(T_INT2*& arg);`
**Remarks:** Extract an INT2_ARRAY field from a message.
**C API:** TipcMsgNextInt2Array

**Syntax:** `virtual TipcMsg& operator>>(T_INT4& arg);`

**Remarks:** Extract an INT4 field from a message.

**C API:** TipcMsgNextInt4

**Syntax:** `virtual TipcMsg& operator>>(T_INT4*& arg);`

**Remarks:** Extract an INT4_ARRAY field from a message.

**C API:** TipcMsgNextInt4Array

**Syntax:** `virtual TipcMsg& operator>>(T_INT8& arg);`

**Remarks:** Extract an INT8 field from a message.

**C API:** TipcMsgNextInt8

**Syntax:** `virtual TipcMsg& operator>>(T_INT8*& arg);`

**Remarks:** Extract an INT8_ARRAY field from a message.

**C API:** TipcMsgNextInt8Array

**Syntax:** `virtual TipcMsg& operator>>(TipcMsg& arg);`

**Remarks:** Extract a T_IPC_MSG field from a message and assign it to the TipcMsg object argument of the operator.

**C API:** TipcMsgNextMsg

**Syntax:** `virtual TipcMsg& operator>>(TipcMsg*& arg);`

**Remarks:** Extract a MSG_ARRAY field from a message and assign each element of the array to an element of a TipcMsg object array.

**C API:** TipcMsgNextMsgArray

**Syntax:** `virtual TipcMsg& operator>>(T_REAL4& arg);`

**Remarks:** Extract a REAL4 field from a message.

**C API:** TipcMsgNextReal4

**Syntax:** `virtual TipcMsg& operator>>(T_REAL4*& arg);`

**Remarks:** Extract a REAL4_ARRAY field from a message.

**C API:** TipcMsgNextReal4Array

**Syntax:** `virtual TipcMsg& operator>>(T_REAL8& arg);`

**Remarks:** Extract a REAL8 field from a message.

**C API:** TipcMsgNextReal8

**Syntax:** `virtual TipcMsg& operator>>(T_REAL8*& arg);`

**Remarks:** Extract a REAL8_ARRAY field from a message.

**C API:** TipcMsgNextStrArray

**Syntax:** `virtual TipcMsg& operator>>(T_REAL16& arg);`

**Remarks:** Extract a REAL16 field from a message.

**C API:** TipcMsgNextReal16

**Syntax:** `virtual TipcMsg& operator>>(T_REAL16*& arg);`

**Remarks:** Extract a REAL16_ARRAY field from a message.

**C API:** TipcMsgNextReal16Array

**Syntax:** `virtual TipcMsg& operator>>(T_STR& arg);`

**Remarks:** Extract a STR field from a message.

**C API:** TipcMsgNextStr

**Syntax:** `virtual TipcMsg& operator>>(T_STR*& arg);`

**Remarks:** Extract a STR_ARRAY field from a message.

**C API:** TipcMsgNextStrArray

**Syntax:** `virtual TipcMsg& operator>>(T_XML*& arg);`

**Remarks:** Extract an XML field from a message.

**C API:** TipcMsgNextXml

## Manipulators

### TipcMsg::Check

**Syntax:** `TipcMsg& Check(TipcMsg& msg);`

**Remarks:** Data fields can be inserted into or extracted from a TipcMsg object by chaining insertion (operator<<) and extraction (operator>>) operators. If any particular insertion or extraction operator fails, a problem flag is set inside the TipcMsg object to indicate that an error has occurred. The problem flag stays set until it is reset using TipcMsg::CheckReset(). When the Check manipulator is used in a function chain, it sets the Tobj::_status flag of the TipcMsg object to FALSE whenever the problem flag is set. Likewise, if the problem flag is unset, then the Check manipulator sets the object's Tobj::_status flag to TRUE.

After the Check manipulator has been invoked, normal methods of accessing an object's status can be employed.

## Parameterized Manipulators

### TipcMsg::GetSize

**Syntax:** `TipcMsgManip GetSize(T_INT4 *size_return);`

**Remarks:** This manipulator returns the size of an array or binary data previously extracted using the TipcMsg extraction operator (operator>>). The size is returned in *size_return*.

### TipcMsg::SetSize

**Syntax:** `TipcMsgManip SetSize(T_INT4 size);`

**Remarks:** This manipulator informs the TipcMsg object of the *size* of the next array or binary data to be inserted into the object.

## Static Public Member Functions

### TipcMsg::FieldSize

**Syntax:** `static T_BOOL FieldSize(T_IPC_MSG_FIELD field, T_INT4 size);`

**Remarks:** Set the size of a message field.

**C API:** TipcMsgFieldSetSize

### TipcMsg::UpdatePtr

**Syntax:** `T_BOOL UpdatePtr(T_IPC_MSG_FIELD field, T_PTR binary_data, T_INT4 array_size);`

**Remarks:** Update the pointer of the MSG field returned by TipcMsgAppendBinaryPtr.

**C API:** TipcMsgFieldUpdateBinaryPtr

**Syntax:** `T_BOOL UpdateBoolPtr(T_IPC_MSG_FIELD field, T_BOOL *bool_array, T_INT4 array_size);`

**Remarks:** Update the pointer of the MSG field returned by TipcMsgAppendBoolArrayPtr.

**C API:** TipMsgFieldUpdateBoolArrayPtr

**Syntax:** `T_BOOL UpdatePtr(T_IPC_MSG_FIELD field, T_INT2 *int2_array, T_INT4 array_size);`

**Remarks:** Update the pointer of the MSG field returned by TipcMsgAppendInt2ArrayPtr.

**C API:** TipMsgFieldUpdateInt2ArrayPtr

**Syntax:** `T_BOOL UpdatePtr(T_IPC_MSG_FIELD field, T_INT4 *int4_array, T_INT4 array_size);`

**Remarks:** Update the pointer of the MSG field returned by TipcMsgAppendInt4ArrayPtr.

**C API:** TipMsgFieldUpdateInt4ArrayPtr

**Syntax:** `T_BOOL UpdatePtr(T_IPC_MSG_FIELD field, T_INT8 *int8_array, T_INT4 array_size);`

**Remarks:** Update the pointer of the MSG field returned by TipcMsgAppendInt8ArrayPtr.

**C API:** TipMsgFieldUpdateInt8ArrayPtr

**Syntax:** `T_BOOL UpdatePtr(T_IPC_MSG_FIELD` *field*`, T_REAL4 *`*real4_array*`, T_INT4` *array_size*`);`

**Remarks:** Update the pointer of the MSG field returned by TipcMsgAppendReal4ArrayPtr.

**C API:** TipMsgFieldUpdateReal4ArrayPtr

**Syntax:** `T_BOOL UpdatePtr(T_IPC_MSG_FIELD` *field*`, T_REAL8 *`*real8_array*`, T_INT4` *array_size*`);`

**Remarks:** Update the pointer of the MSG field returned by TipcMsgAppendReal8ArrayPtr.

**C API:** TipMsgFieldUpdateReal8ArrayPtr

**Syntax:** `T_BOOL UpdatePtr(T_IPC_MSG_FIELD` *field*`, T_REAL16 *`*real16_array*`, T_INT4` *array_size*`);`

**Remarks:** Update the pointer of the MSG field returned by TipcMsgAppendReal16ArrayPtr.

**C API:** TipMsgFieldUpdateReal16ArrayPtr

**Syntax:** `T_BOOL UpdatePtr(T_IPC_MSG_FIELD` *field*`, T_STR *`*str_array_data*`, T_INT4` *array_size*`);`

**Remarks:** Update the pointer of the MSG field returned by TipcMsgAppendStrArrayPtr.

**C API:** TipMsgFieldUpdateStrArrayPtr

**Syntax:** `T_BOOL UpdatePtr(T_IPC_MSG_FIELD` *field*`, T_STR` *str_data*`);`

**Remarks:** Update the pointer of the MSG field returned by TipcMsgAppendStrPtr.

**C API:** TipMsgFieldUpdateStrPtr

**Syntax:** `T_BOOL UpdatePtr(T_IPC_MSG_FIELD` *field*`, T_IPC_MSG *`*msg_array_data*`, T_INT4` *array_size*`);`

**Remarks:** Update the pointer of the MSG field returned by TipcMsgAppendMsgArrayPtr.

**C API:** TipMsgFieldUpdateMsgArrayPtr

**Syntax:** `T_BOOL UpdatePtr(T_IPC_MSG_FIELD field, T_IPC_MSG msg_data);`

**Remarks:** Update the pointer of the MSG field returned by TipcMsgAppendMsgPtr.

**C API:** TipMsgFieldUpdateMsgPtr

**Syntax:**
```
T_BOOL UpdatePtrUtf8(T_IPC_MSG_FIELD field,
                     T_STR *utf8_array_data,
                     T_INT4 array_size);
```

**Remarks:** Update the pointer of the MSG field returned by TipcMsgAppendUtf8ArrayPtr.

**C API:** TipcMsgFieldUpdateUtf8ArrayPtr

**Syntax:** `T_BOOL UpdatePtrUtf8(T_IPC_MSG_FIELD field, T_STR utf8_data);`

**Remarks:** Update the pointer of the MSG field returned by TipcMsgAppendUtf8Ptr.

**C API:** TipcMsgFieldUpdateUtf8Ptr

**Syntax:** `T_BOOL UpdatePtr(T_IPC_MSG_FIELD field, T_XML xml_data);`

**Remarks:** Update the pointer of the MSG field returned by TipcMsgAppendXmlPtr.

**C API:** TipcMsgFieldUpdateXmlPtr

### TipcMsg::FtToStr

**Syntax:** `static T_BOOL FtToStr(T_IPC_FT type, T_STR *str_return);`

**Remarks:** Convert a field type to a string.

**C API:** TipcFtToStr

### TipcMsg::LbModeToStr

**Syntax:**
```
static T_BOOL LbModeToStr(T_IPC_LB_MODE lb_mode,
                          T_STR *str_return);
```

**Remarks:** Convert a load balancing mode to a string for printing.

**C API:** TipcLbModeToStr

### TipcMsg::DeliveryModeToStr

**Syntax:** `static T_BOOL DeliveryModeToStr(T_IPC_DELIVERY_MODE` *delivery_mode*`, T_STR *`*delivery_mode_str_return*`);`

**Remarks:** Convert a delivery mode to a string.

**C API:** TipcDeliveryModeToStr

### TipcMsg::StrToFt

**Syntax:** `static T_BOOL StrToFt(T_STR` *str*`, T_IPC_FT *`*ft_return*`);`

**Remarks:** Convert a string to a field type.

**C API:** TipcStrToFt

### TipcMsg::StrToLbMode

**Syntax:** `static T_BOOL StrToLbMode(T_STR` *str*`, T_IPC_LB_MODE *`*lb_mode_return*`);`

**Remarks:** Convert a string into a load balancing mode.

**C API:** TipcStrToLbMode

### TipcMsg::StrToDeliveryMode

**Syntax:** `static T_BOOL StrToDeliveryMode(T_STR` *delivery_str*`, T_IPC_DELIVERY_MODE *`*deliv_mode_return*`);`

**Remarks:** Convert a string to a delivery mode.

**C API:** TipcStrToDeliveryMode

## Protected Member Functions

### TipcMsg::size

**Syntax:** `T_INT4 size() const;`

**Remarks:** Get the `_size` field of the TipcMsg object used to insert or extract array and BINARY data fields.

**Syntax:** `void size(T_INT4` *size_arg*`);`

**Remarks:** Set the `_size` field of the TipcMsg object used to insert or extract array and BINARY data fields.

## Example

This code fragment shows how to construct a NUMERIC_DATA message, set some of its values, and then access this data:

```
#include <rtworks/cxxipc.hxx>

int main()
{
  cout << "Create the message." << endl;
  TipcMsg msg(T_MT_NUMERIC_DATA);
  if (!msg) {
    // error
  }
  cout << "Set the message properties." << endl;
  msg.Sender("_conan");
  if (!msg) {
    // error
  }

  msg.Dest("thermal");
  if (!msg) {
    // error
  }

  msg.Priority(2);
  if (!msg) {
    // error
  }

  cout << "Append fields." << endl;
  msg << "voltage" << (T_REAL8)33.4534
      << "switch_pos" << (T_REAL8)0.0 << Check;

  if (!msg) {
    // error
  }

  cout << "Access fields." << endl;

  msg.Current(0);
  if (!msg) {
    // error
  }
    T_STR str_val;
  msg >> str_val;
  if (!msg) {
    // error
  }
  T_REAL8 real8_val;
  msg >> real8_val;
  if (!msg) {
    // error
  }
  cout << str_val << " = " << real8_val << endl;
    msg >> str_val;
```

```
 if (!msg) {
// error
 }
 msg >> real8_val;
 if (!msg) {
   // error
 }

 cout << str_val << " = " << real8_val << endl;
 cout << "Destroy the message." << endl;

 return T_EXIT_SUCCESS; // all done
// main
```

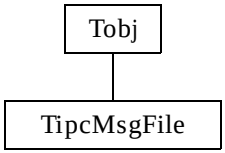
# TipcMsgFile

**Name** TipcMsgFile — user class for message file I/O operations

**Synopsis**

```
TipcMsgFile msg_file(file_name, create_mode);
TipcMsgFile msg_file(file, create_mode);
```

**Inheritance**




**Description** The TipcMsgFile class provides a C++ binding to the SmartSockets TipcMsgFile API functions using a virtual I/O stream model. It allows you to create, read, write, append to and close SmartSockets message files using C++ stream I/O syntax.

**Caution** None

## Construction/Destruction

### TipcMsgFile::TipcMsgFile

**Syntax:**
```
TipcMsgFile(T_STR file_name,
            T_IPC_MSG_FILE_CREATE_MODE create_mode);
```

**Remarks:** Create a TipcMsgFile object with *file_name* and *create_mode*. *file_name* is the name of the file to use for the message file. *create_mode* specifies how to create the file.

**C API:** TipcMsgFileCreate

**Syntax:**
```
TipcMsgFile(FILE *file,
            T_IPC_MSG_FILE_CREATE_MODE create_mode);
```

**Remarks:** Create a TipcMsgFile object with *file* and *create_mode*. *file* is an existing file to use for the message file. *create_mode* specifies how to create (that is, open) the file.

**C API:** TipcMsgFileCreateFromFile

### TipcMsgFile::~TipcMsgFile

**Syntax:** `~TipcMsgFile();`

**Remarks:** Destroy a TipcMsgFile object. The destructor calls the C API function TipcMsgFileDestroy.

**C API:** TipcMsgFileDestroy

## Operators

### TipcMsgFile::operator>>

**Syntax:** `virtual TipcMsgFile& operator>>(TipcMsg& msg);`

**Remarks:** Read a message from a message file and assign it to the internal T_IPC_MSG field of a TipcMsg object. Any message that the object was managing at the time of the call is passed to TipcMsgDestroy().

**C API:** TipcMsgFileRead

### TipcMsgFile::operator<<

**Syntax:** `virtual TipcMsgFile& operator<<(TipcMsg& msg);`

**Remarks:** Write the internal T_IPC_MSG field of a TipcMsg object to a message file.

**C API:** TipcMsgFileWrite

## Example

This example code fragment creates a file stream from the file `data.msg`, reads all the messages in the file using the extraction operator, and prints them out:

```
#include <rtworks/cxxipc.hxx>
int main()
{
  TipcMsgFile msg_file("data.msg", T_IPC_MSG_FILE_CREATE_READ);
  if (!msg_file) {
   // error
  }

TipcMsg msg_read;
  while (1) {
   msg_file >> msg_read;
    if (!msg_file) {
    break;
  }
```

```
  TutOut("Read a message.\n");

  msg_read.Print(TutOut);
  }
  // Note: file stream automatically destroyed
}
```

If the file `data.msg` contains this information:

```
numeric_data thermal {
 voltage 33.4534
 switch_pos -8843.8
}
```

The output is similar to this:

```
Read a message.
type = numeric_data
sender = <_file>
dest = <thermal>
max = 2048
size = 75
current = 0
read_only = false
priority = 0
delivery_mode = best_effort
ref_count = 1
seq_num = 0
resend_mode = false
user_prop = 0
data (num_fields = 4):
  str "voltage"
  real8 33.4534
  str "switch_pos"
  real8 -8843.8
```

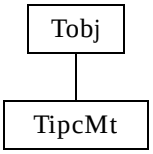
# TipcMt

**Name** TipcMt — user class for constructing, looking up, and destroying message types

**Synopsis**

```
TipcMt mt(mt_num);
TipcMt mt(mt_name);
TipcMt mt(name, num, grammar);
TipcMt mt(mt);
```

**Inheritance**




**Description** The TipcMt class is a user C++ interface to the TipcMt-prefix SmartSockets function calls.

**Caution** None

**See Also** TipcMsg, Tobj

## Construction/Destruction

### TipcMt::TipcMt

**Syntax:** `TipcMt();`

**Remarks:** Create a vacant TipcMt object.

**Syntax:** `TipcMt(T_INT4 mt_num);`

**Remarks:** Create a TipcMt object using *mt_num*. *mt_num* is the number of the message type used to create the object. This constructor uses the C API TipcMtLookupByNum to initialize the object's internal T_IPC_MT message type field.

**C API:** TipcMtLookupByNum

**Syntax:** `TipcMt(T_STR mt_name);`

**Remarks:** Create a TipcMt object using *mt_name*. *mt_name* is the name of a message type (not case sensitive). This constructor uses the C API TipcMtLookup to initialize the object's internal T_IPC_MT message type field.

**C API:** TipcMtLookup

**Syntax:** `TipcMt(T_STR name, T_INT4 num, T_STR grammar);`

**Remarks:** Create a TipcMt object using *name*, *num*, and *grammar*. This constructor is used to create a new message type. *name* is the name of the new message type (not case sensitive). *num* is the number associated with the new message type. *grammar* is the description for reading and writing message files. This constructor employs the C API's TipcMtCreate to create a new message type and initialize the object's internal T_IPC_MT message type field.

**C API:** TipcMtCreate

**Syntax:** `TipcMt(T_IPC_MT mt);`

**Remarks:** Create a TipcMt object using *mt*. *mt* is an initialized C API T_IPC_MT type.

**Syntax:** `TipcMt(const TipcMt& mt);`

**Remarks:** Create a TipcMt object using *mt*. *mt* is another TipcMt object. This constructor is known as the TipcMt copy constructor.

### TipcMt::~TipcMt

**Syntax:** `~TipcMt();`

**Remarks:** This destructor just destroys the object and not the message type to which it referred.

## Public Member Functions

### TipcMt::Create

**Syntax:** `T_BOOL Create(T_STR name, T_INT4 num, T_STR grammar);`

**Remarks:** Create a user-defined message type using *name*, *num*, and *grammar*. This member function is used to create a new message type. *name* is the name of the new message type (not case sensitive). *num* is the number associated with the new message type. *grammar* is the description for reading and writing message files. This constructor employs the C API's TipcMtCreate to create a new message type and initialize the object's internal T_IPC_MT message type field.

**C API:** TipcMtCreate

### TipcMt::DeliveryMode

**Syntax:** `T_IPC_DELIVERY_MODE DeliveryMode();`

**Remarks:** Get the delivery mode of a message type.

**C API:** TipcMtGetDeliveryMode

**Syntax:** `T_BOOL DeliveryMode(T_IPC_DELIVERY_MODE delivery_mode);`

**Remarks:** Set the delivery mode of a message type.

**C API:** TipcMtSetDeliveryMode

### TipcMt::DeliveryTimeout

**Syntax:** `T_REAL8 DeliveryTimeout();`

**Remarks:** Get the delivery timeout of a message type.

**C API:** TipcMtGetDeliveryTimeout

**Syntax:** `T_BOOL DeliveryTimeout(T_REAL8 delivery_timeout);`

**Remarks:** Set the delivery timeout of a message type.

**C API:** TipcMtSetDeliveryTimeout

### TipcMt::Destroy

**Syntax:** `T_BOOL Destroy();`

**Remarks:** Destroy a message type.

**C API:** TipcMtDestroy

### TipcMt::Grammar

**Syntax:** `T_STR Grammar();`

**Remarks:** Get the grammar of a message type.

**C API:** TipcMtGetGrammar

### TipcMt::HeaderStrEncode

**Syntax:** `T_BOOL HeaderStrEncode();`

**Remarks:** Get the header string encoding mode for a message type.

**C API:** TipcMtGetHeaderStrEncode

**Syntax:** `T_BOOL HeaderStrEncode(T_BOOL` *header_str_encode*`);`

**Remarks:** Set the header string encoding mode for a message type.

**C API:** TipcMtSetHeaderStrEncode

### TipcMt::LbMode

**Syntax:** `T_IPC_LB_MODE LbMode();`

**Remarks:** Get the load balancing mode of a message type.

**C API:** TipcMtGetLbMode

**Syntax:** `T_BOOL LbMode(T_IPC_LB_MODE` *lb_mode*`);`

**Remarks:** Set the load balancing mode of a message type.

**C API:** TipcMtSetLbMode

### TipcMt::Lookup

**Syntax:** `T_BOOL Lookup(T_INT4 num);`

**Remarks:** Look up a message type by number. This member function sets the object's internal T_IPC_MT message type field.

**C API:** TipcMtLookupByNum

**Syntax:** `T_BOOL Lookup(T_STR name):`

**Remarks:** Look up a message type by name. This member function sets the object's internal T_IPC_MT message type field.

**C API:** TipcMtLookup

### TipcMt::MessageType

**Syntax:** `T_IPC_MT MessageType() const;`

**Remarks:** Return the internal T_IPC_MT message type of the object.

### TipcMt::Name

**Syntax:** `T_STR Name();`

**Remarks:** Get the name of a message type.

**C API:** TipcMtGetName

### TipcMt::Num

**Syntax:** `T_INT4 Num();`

**Remarks:** Get the number of a message type.

**C API:** TipcMtGetNum

### TipcMt::Print

**Syntax:** `T_BOOL Print(T_OUT_FUNC func);`

**Remarks:** Print all information about a message type.

**C API:** TipcMtPrint

### TipcMt::Priority

**Syntax:** `T_INT2 Priority();`

**Remarks:** Get the priority of a message type.

**C API:** TipcMtGetPriority

**Syntax:** `T_BOOL Priority(T_INT2 priority);`

**Remarks:** Set the priority of a message type.

**C API:** TipcMtSetPriority

### TipcMt::SetPriorityUnknown

**Syntax:** `T_BOOL SetPriorityUnknown();`

**Remarks:** Set the priority of a message type to unknown.

**C API:** TipcMtSetPriorityUnknown

### TipcMt::UserProp

**Syntax:** `T_INT4 UserProp();`

**Remarks:** Get user-defined property from a message type.

**C API:** TipcMtGetUserProp

**Syntax:** `T_BOOL UserProp(T_INT4 user_prop);`

**Remarks:** Set user-defined property of a message type.

**C API:** TipcMtSetUserProp

## Conversion Operators

### TipcMt::T_IPC_MT

**Syntax:** `operator T_IPC_MT() const;`

**Remarks:** Convert a TipcMt object to a C API T_IPC_MT type by returning the internal T_IPC_MT message type.

## Static Public Member Functions

### TipcMt::Traverse

**Syntax:** `static T_PTR Traverse(T_IPC_MT_TRAV_FUNC` *func*`, T_PTR` *arg*`);`

**Remarks:** Traverse all message types.

**C API:** TipcMtTraverse

## Example

This example creates a message type named "xyz_coord" that uses three integer coordinates, creates an XYZ_COORD message, appends fields, and writes the message to a message file. See the description of the TipcMtCreate function in the *TIBCO SmartSockets Application Programming Interface* for a corresponding C language version of this code fragment.

```
#include <rtworks/cxxipc.hxx>
main()
{
#define USER_MT_XYZ_COORD 2000

 // Create the message type
   TipcMt mt;
   mt.Create("xyz_coord", USER_MT_XYZ_COORD, "int4 int4 int4");

 // Create a message and append fields
   TipcMsg msg(mt);

   msg << (T_INT4)5 << (T_INT4)2 << (T_INT4)9;

 // Create a message file
   TipcMsgFile msg_file("test.msg", T_IPC_MSG_FILE_CREATE_WRITE);

   if (!msg_file) {
  // error
   }

 // Write the message out
   msg_file << msg.Message();

}
```

The program writes this to the message file `test.msg`:

```
xyz_coord _null 5 2 9
```

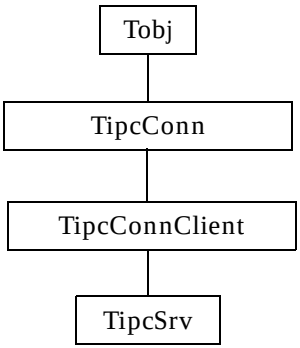
# TipcSrv

**Name** TipcSrv — user class representing the RTclient functionality (corresponding to the TipcSrv* API functions)

**Synopsis**

```
static TipcSrv& Instance();
static TipcSrv& InstanceCreate();
static TipcSrv& InstanceCreate(create_status);
```

**Inheritance**




**Description** The TipcSrv class provides a C++ interface to the TipcSrv* functions in SmartSockets.

The TipcSrv constructor is protected because in SmartSockets you cannot have more than one RTserver connection per process. Therefore, you are not allowed to construct a TipcSrv instance directly. If this is attempted, the corresponding C++ code does not pass compilation. However, you still need to be able to get a handle to a TipcSrv instance in order to call the TipcSrv* versions of TipcConn* virtual functions inherited from the TipcConn class, and you still need to be able to cause TipcSrvCreate() to be called by some mechanism. For the underlying C function TipcSrvCreate() to be called, you should instead call the static TipcSrv class function InstanceCreate() or Instance(), the difference between these functions being whether TipcSrvCreate gets called or not as a side effect. Because C++ doesn't allow arguments to destructors, the TipcSrv class offers a Destroy() method to let you specify a SmartSockets destroy status. When TipcSrv is destroyed automatically (for example, Destroy() isn't called, and the TipcSrv reference obtained through Instance() passes out of scope), the T_IPC_SRV_CONN_NONE destroy status is set by default.

**Caution** None

**See Also** TipcConn

## Public Member Functions

### TipcSrv::Arch

**Syntax:** `virtual T_STR Arch();`

**Remarks:** Get the architecture name of the connected RTserver.

**C API:** TipcSrvGetArch

### TipcSrv::AutoFlushSize

**Syntax:** `virtual T_INT4 AutoFlushSize();`

**Remarks:** Get the automatic flush size of a connection to RTserver.

**C API:** TipcSrvGetAutoFlushSize

**Syntax:** `virtual T_BOOL AutoFlushSize(T_INT4 auto_flush_size);`

**Remarks:** Set the automatic flush size of a connection to RTserver.

**C API:** TipcSrvSetAutoFlushSize

### TipcSrv::BlockMode

**Syntax:** `virtual T_BOOL BlockMode();`

**Remarks:** Get the block mode of the connection to RTserver.

**C API:** TipcSrvGetBlockMode

**Syntax:** `virtual T_BOOL BlockMode(T_BOOL block_mode);`

**Remarks:** Set the block mode of the connection to RTserver.

**C API:** TipcSrvSetBlockMode

### TipcSrv::Check

**Syntax:**
```
virtual T_BOOL Check(T_IO_CHECK_MODE check_mode,
                     T_REAL8 timeout);
```

**Remarks:** Check if data can be read from or written to the connection to RTserver.

**C API:** TipcSrvCheck

### TipcSrv::ConnStatus

**Syntax:** `T_IPC_SRV_CONN_STATUS ConnStatus();`

**Remarks:** Get the status of the connection to RTserver.

**C API:** TipcSrvGetConnStatus

### TipcSrv::Create

**Syntax:** `T_BOOL Create(T_IPC_SRV_CONN_STATUS` *create_status*`);`

**Remarks:** Create the connection to RTserver.

**C API:** TipcSrvCreate

### TipcSrv::DefaultCbCreate

**Syntax:**
```
virtual T_CB
DefaultCbCreate(T_IPC_CONN_DEFAULT_CB_FUNC func,
                T_CB_ARG arg);
```

**Remarks:** Create a default callback in the connection to RTserver.

**C API:** TipcSrvDefaultCbCreate

### TipcSrv::DefaultCbLookup

**Syntax:**
```
virtual T_CB
DefaultCbLookup(T_IPC_CONN_DEFAULT_CB_FUNC func,
                T_CB_ARG arg);
```

**Remarks:** Look up a default callback in the connection to RTserver.

**C API:** TipcSrvDefaultCbLookup

### TipcSrv::ErrorCbCreate

**Syntax:**
```
virtual T_CB ErrorCbCreate(T_IPC_CONN_ERROR_CB_FUNC func,
                           T_CB_ARG arg);
```

**Remarks:** Create an error callback in the connection to RTserver.

**C API:** TipcSrvErrorCbCreate

### TipcSrv::ErrorCbLookup

**Syntax:** `virtual T_CB ErrorCbLookup(T_IPC_CONN_ERROR_CB_FUNC` *`func`*`, T_CB_ARG` *`arg`*`);`

**Remarks:** Look up an error callback in the connection to RTserver.

**C API:** TipcSrvErrorCbLookup

### TipcSrv::Flush

**Syntax:** `virtual T_BOOL Flush();`

**Remarks:** Flush buffered outgoing messages on the connection to RTserver.

**C API:** TipcSrvFlush

### TipcSrv::GmdFileCreate

**Syntax:** `virtual T_BOOL GmdFileCreate();`

**Remarks:** Create guaranteed message delivery area on the connection to RTserver.

**C API:** TipcSrvGmdFileCreate

### TipcSrv::GmdFileDelete

**Syntax:** `virtual T_BOOL GmdFileDelete();`

**Remarks:** Delete guaranteed message delivery files for the connection to RTserver.

**C API:** TipcSrvGmdFileDelete

### TipcSrv::GmdMaxSize

**Syntax:** `T_UINT4 GmdMaxSize();`

**Remarks:** Get the GMD area maximum size of the connection to RTserver.

**C API:** TipcSrvGetGmdMaxSize

**Syntax:** `T_BOOL GmdMaxSize(T_UINT4` *`gmd_max_size`*`);`

**Remarks:** Set the GMD area maximum size of the connection to RTserver.

**C API:** TipcSrvSetGmdMaxSize

### TipcSrv::GmdMsgDelete

**Syntax:** `T_BOOL GmdMsgDelete(TipcMsg& msg);`

**Remarks:** Delete a message from the GMD area after a GMD failure on the connection to RTserver.

**C API:** TipcSrvGmdMsgDelete

### TipcSrv::GmdMsgResend

**Syntax:** `T_BOOL GmdMsgResend(TipcMsg& msg);`

**Remarks:** Resend a message after a GMD failure on the connection to RTserver.

**C API:** TipcSrvGmdMsgResend

### TipcSrv::GmdMsgServerDelete

**Syntax:** `T_BOOL GmdMsgServerDelete(TipcMsg& msg);`

**Remarks:** Delete a message in RTserver after a GMD failure on the connection to RTserver.

**C API:** TipcSrvGmdMsgServerDelete

### TipcSrv::GmdMsgStatus

**Syntax:** `T_BOOL GmdMsgStatus(TipcMsg& msg);`

**Remarks:** Poll RTserver for GMD status of a message.

**C API:** TipcSrvGmdMsgStatus

### TipcSrv::GmdNumPending

**Syntax:** `T_INT4 GmdNumPending();`

**Remarks:** Get the number of guaranteed messages pending on connection to RTserver.

**C API:** TipcSrvGetGmdNumPending

### TipcSrv::GmdResend

**Syntax:** `virtual T_BOOL GmdResend();`

**Remarks:** Resend all guaranteed messages after a delivery failure on the connection to RTserver.

**C API:** TipcSrvGmdResend

### TipcSrv::Insert

**Syntax:** `virtual T_BOOL Insert(TipcMsg& msg, T_INT4 pos);`

**Remarks:** Insert a message into queue of the connection to RTserver.

**C API:** TipcSrvMsgInsert

### TipcSrv::KeepAlive

**Syntax:** `virtual T_BOOL KeepAlive();`

**Remarks:** Check if the connection to RTserver is still alive.

**C API:** TipcSrvKeepAlive

### TipcSrv::Lock

**Syntax:** `virtual T_BOOL Lock();`

**Remarks:** Acquire exclusive access to the connection to RTserver.

**C API:** TipcSrvLock

### TipcSrv::MainLoop

**Syntax:** `virtual T_BOOL MainLoop(T_REAL8 timeout);`

**Remarks:** Read and process messages on the connection to RTserver.

**C API:** TipcSrvMainLoop

### TipcSrv::Next

**Syntax:** `virtual TipcMsg& Next(T_REAL8 timeout);`

**Remarks:** Get the next message from the connection to RTserver.

**C API:** TipcSrvMsgNext

### TipcSrv::Node

**Syntax:** `virtual T_STR Node();`

**Remarks:** Get the node name of the connected RTserver.

**C API:** TipcSrvGetNode

### TipcSrv::NumQueued

**Syntax:** `virtual T_INT4 NumQueued();`

**Remarks:** Get number of queued messages from the connection to RTserver.

**C API:** TipcSrvGetNumQueued

### TipcSrv::Pid

**Syntax:** `virtual Pid();`

**Remarks:** Get the process ID of the connected RTserver.

**C API:** TipcSrvGetPid

### TipcSrv::Print

**Syntax:** `virtual Print(T_OUT_FUNC` *`func`*`);`

**Remarks:** Print all information about the connection to RTserver.

**C API:** TipcSrvPrint

### TipcSrv::Process

**Syntax:** `virtual T_BOOL Process(TipcMsg&` *`msg`*`);`

**Remarks:** Process a message in the connection to RTserver.

**C API:** TipcSrvMsgProcess

### TipcSrv::ProcessCbCreate

**Syntax:**
```
virtual T_CB ProcessCbCreate(TipcMt& mt,
                             T_IPC_CONN_PROCESS_CB_FUNC func,
                             T_CB_ARG arg);
```

**Remarks:** Create a process callback in the connection to RTserver.

**C API:** TipcSrvProcessCbCreate

### TipcSrv::ProcessCbLookup

**Syntax:**
```
virtual T_CB ProcessCbLookup(TipcMt& mt,
                             T_IPC_CONN_PROCESS_CB_FUNC func,
                             T_CB_ARG arg);
```

**Remarks:** Look up a process callback in the connection to RTserver.

**C API:** TipcSrvProcessCbLookup

### TipcSrv::QueueCbCreate

**Syntax:**
```
virtual T_CB QueueCbCreate(TipcMt& mt,
                           T_IPC_CONN_QUEUE_CB_FUNC func,
                           T_CB_ARG arg);
```

**Remarks:** Create a queue callback in the connection to RTserver.

**C API:** TipcSrvQueueCbCreate

### TipcSrv::QueueCbLookup

**Syntax:**
```
virtual T_CB QueueCbLookup(TipcMt& mt,
                           T_IPC_CONN_QUEUE_CB_FUNC func,
                           T_CB_ARG arg);
```

**Remarks:** Look up a queue callback in the connection to RTserver.

**C API:** TipcSrvQueueCbLookup

### TipcSrv::Read

**Syntax:**
```
virtual T_BOOL Read(T_REAL8 timeout);
```

**Remarks:** Read all available data from the connection to RTserver and queue messages in priority order.

**C API:** TipcSrvRead

### TipcSrv::ReadCbCreate

**Syntax:**
```
virtual T_CB ReadCbCreate(TipcMt& mt,
                          T_IPC_CONN_READ_CB_FUNC func,
                          T_CB_ARG arg);
```

**Remarks:** Create a read callback in the connection to RTserver.

**C API:** TipcSrvReadCbCreate

### TipcSrv::ReadCbLookup

**Syntax:** `virtual T_CB ReadCbLookup(TipcMt& mt, T_IPC_CONN_READ_CB_FUNC func, T_CB_ARG arg);`

**Remarks:** Look up a read callback in the connection to RTserver.

**C API:** TipcSrvReadCbLookup

### TipcSrv::Search

**Syntax:** `virtual TipcMsg& Search(T_REAL8 timeout, T_IPC_CONN_MSG_SEARCH_FUNC func, T_PTR arg);`

**Remarks:** Search the message queue of the connection to RTserver for a specific message.

**C API:** TipcSrvMsgSearch

### TipcSrv::SearchType

**Syntax:** `virtual TipcMsg& SearchType(T_REAL8 timeout, TipcMt& mt);`

**Remarks:** Search the message queue of the connection to RTserver for a message with a specific type.

**C API:** TipcSrvMsgSearchType

### TipcSrv::Send

**Syntax:** `virtual T_BOOL Send(TipcMsg& msg, T_BOOL check_server_msg_send = FALSE);`

**Remarks:** Send a message through the connection to RTserver.

**C API:** TipcSrvMsgSend

### TipcSrv::SendRpc

**Syntax:** `virtual TipcMsg& SendRpc(TipcMsg& call_msg, T_REAL8 timeout);`

**Remarks:** Make a remote procedure call (RPC) with a message on the connection to RTserver.

**C API:** TipcSrvMsgSendRpc

### TipcSrv::Socket

**Syntax:** `virtual T_INT4 Socket();`

**Remarks:** Get the socket of the connection to RTserver.

**C API:** TipcSrvGetSocket

**Syntax:** `virtual T_BOOL Socket(T_INT4 socket);`

**Remarks:** Set the socket of the connection to RTserver.

**C API:** TipcSrvSetSocket

### TipcSrv::SubjectCbCreate

**Syntax:**
```
virtual T_CB SubjectCbCreate(T_STR subject, T_IPC_MT mt,
                             T_IPC_SRV_SUBJECT_FUNC func,
                             T_CB_ARG arg);
```

**Remarks:** Create a subject callback.

**C API:** TipcSrvSubjectCbCreate

### TipcSrv::SubjectCbLookup

**Syntax:**
```
virtual T_CB SubjectCbLookup(T_STR subject, T_IPC_MT mt,
                             T_IPC_SRV_SUBJECT_FUNC func,
                             T_CB_ARG arg);
```

**Remarks:** Look up a subject callback.

**C API:** TipcSrvSubjectCbLookup

### TipcSrv::SubjectDefaultCbCreate

**Syntax:**
```
T_CB SubjectDefaultCbCreate(T_IPC_SRV_SUBJECT_FUNC func,
                            T_CB_ARG arg);
```

**Remarks:** Create the default subject callback.

**C API:** TipcSrvSubjectDefaultCbCreate

### TipcSrv::SubjectDefaultCbLookup

**Syntax:**
```
T_CB SubjectDefaultCbLookup(T_IPC_SRV_SUBJECT_FUNC func,
                            T_CB_ARG arg);
```

**Remarks:** Look up the default subject callback.

**C API:** TipcSrvSubjectDefaultCbLookup

### TipcSrv::SubjectCbDestroyAll

**Syntax:** `virtual T_BOOL SubjectCbDestroyAll();`

**Remarks:** Destroy all subject callbacks.

**C API:** TipcSrvSubjectCbDestroyAll

### TipcSrv::SubjectGmdInit

**Syntax:** `virtual T_BOOL SubjectGmdInit(T_STR` *subject*`);`

**Remarks:** Initialize GMD for a subject.

**C API:** TipcSrvSubjectGmdInit

### TipcSrv::SubjectLbInit

**Syntax:** `virtual T_BOOL SubjectLbInit(T_STR` *subject*`);`

**Remarks:** Initialize load balancing for a subject.

**C API:** TipcSrvSubjectLbInit

### TipcSrv::SubjectSubscribe

**Syntax:** *T_BOOL SubjectSubscribe(T_STR subject);*

**Remarks:** Get whether or not an RTclient is subscribed to a subject.

**C API:** TipcSrvSubjectGetSubscribe

**Syntax:** `T_BOOL SubjectSubscribe(T_STR` *subject*`, T_BOOL` *subscribe_status*`);`

**Remarks:** Start or stop subscribing to a subject.

**C API:** TipcSrvSubjectSetSubscribe

### TipcSrv::SubjectSubscribeLb

**Syntax:**
```
T_BOOL SubjectSubscribeLb(T_STR subject,
                          T_BOOL *lb_status_return);
```

**Remarks:** Get whether or not an RTclient is subscribed to a subject, including load balancing information.

**C API:** TipcSrvSubjectGetSubscribeLb

**Syntax:** `T_BOOL SubjectSubscribeLb(T_STR subject, T_BOOL subscribe_status, T_BOOL lb_status);`

**Remarks:** Start or stop subscribing to a subject, with or without load balancing.

**C API:** TipcSrvSubjectSetSubscribeLb

### TipcSrv::SrvWrite

**Syntax:** `T_BOOL SrvWrite(T_STR dest, TipcMt& mt, T_BOOL check_server_msg_send, ...);`

**Remarks:** Construct a message and send it through the connection to RTserver.

**C API:** TipcSrvMsgWrite

### TipcSrv::SrvWriteVa

**Syntax:** `T_BOOL SrvWriteVa(T_STR dest, TipcMt& mt, T_BOOL check_server_msg_send, va_list var_arg_list);`

**Remarks:** Construct a message and send it through the connection to RTserver (va_list version).

**C API:** TipcSrvMsgWriteVa

### TipcSrv::Timeout

**Syntax:** `virtual T_REAL8 Timeout(T_IPC_TIMEOUT timeout);`

**Remarks:** Get a timeout property of the connection to RTserver.

**C API:** TipcSrvGetTimeout

**Syntax:** `virtual T_BOOL Timeout(T_IPC_TIMEOUT timeout, T_REAL8 value);`

**Remarks:** Set a timeout property of the connection to RTserver.

**C API:** TipcSrvSetTimeout

### TipcSrv::UniqueSubject

**Syntax:** `virtual T_STR UniqueSubject();`

**Remarks:** Get the unique subject of the connected RTserver.

**C API:** TipcSrvGetUniqueSubject

### TipcSrv::Unlock

**Syntax:** `virtual T_BOOL Unlock();`

**Remarks:** Release exclusive access to the connection to RTserver.

**C API:** TipcSrvUnlock

### TipcSrv::User

**Syntax:** `virtual T_STR User();`

**Remarks:** Get the user name of the connected RTserver.

**C API:** TipcSrvGetUser

### TipcSrv::WriteCbCreate

**Syntax:**
```
virtual T_CB
WriteCbCreate(TipcMt& mt, T_IPC_CONN_WRITE_CB_FUNC func,
              T_CB_ARG arg);
```

**Remarks:** Create a write callback in the connection to RTserver.

**C API:** TipcSrvWriteCbCreate

### TipcSrv::WriteCbLookup

**Syntax:**
```
virtual T_CB WriteCbLookup(TipcMt& mt,
                           T_IPC_CONN_WRITE_CB_FUNC func,
                           T_CB_ARG arg);
```

**Remarks:** Look up a write callback in the connection to RTserver.

**C API:** TipcSrvWriteCbLookup

### TipcSrv::XtSource

**Syntax:** `virtual T_INT4 XtSource();`

**Remarks:** Get source suitable for XtAppAddInput from the connection to RTserver.

**C API:** TipcSrvGetXtSource

## Static Public Member Functions

### TipcSrv::CreateCbCreate

**Syntax:** `static T_CB CreateCbCreate(T_IPC_SRV_CREATE_CB_FUNC` *func*`, T_CB_ARG` *arg*`);`

**Remarks:** Create a server create callback.

**C API:** TipcSrvCreateCbCreate

### TipcSrv::CreateCbLookup

**Syntax:** `static T_CB CreateCbLookup(T_IPC_SRV_CREATE_CB_FUNC` *func*`, T_CB_ARG` *arg*`);`

**Remarks:** Look up a server create callback.

**C API:** TipcSrvCreateCbLookup

### TipcSrv::Destroy

**Syntax:** `static T_BOOL Destroy(T_IPC_SRV_CONN_STATUS` *destroy_status*`);`

**Remarks:** Destroy the connection to RTserver.

**C API:** TipcSrvDestroy

### TipcSrv::DestroyCbCreate

**Syntax:** `static T_CB DestroyCbCreate(T_IPC_SRV_DESTROY_CB_FUNC` *func*`, T_CB_ARG` *arg*`);`

**Remarks:** Create a server destroy callback.

**C API:** TipcSrvDestroyCbCreate

### TipcSrv::DestroyCbLookup

**Syntax:** `static T_CB DestroyCbLookup(T_IPC_SRV_DESTROY_CB_FUNC` *func*`, T_CB_ARG` *arg*`);`

**Remarks:** Look up a server destroy callback.

**C API:** TipcSrvDestroyCbLookup

### TipcSrv::GmdDir

**Syntax:** `static T_STR GmdDir();`

**Remarks:** Get name of directory where files are written for guaranteed message delivery.

**C API:** TipcGetGmdDir

### TipcSrv::Instance

**Syntax:** `static TipcSrv& Instance();`

**Remarks:** Return a handle to a TipcSrv object. Do not create a connection to RTserver at this time.

### TipcSrv::InstanceCreate

**Syntax:** `static TipcSrv& InstanceCreate(T_IPC_SRV_CONN_STATUS` *`create_status`*`);`

**Remarks:** Return a handle to a TipcSrv object and create a connection to RTserver.

### TipcSrv::IsRunning

**Syntax:** `static T_BOOL IsRunning();`

**Remarks:** Check if RTserver is up and running.

**C API:** TipcSrvIsRunning

### TipcSrv::LogAddMt

**Syntax:**
```
static T_BOOL LogAddMt(T_IPC_SRV_LOG_TYPE log_type,
                       TipcMt& mt);
```

**Remarks:** Add a message type to a message file logging type.

**C API:** TipcSrvLogAddMt

### TipcSrv::LogRemoveMt

**Syntax:**
```
static T_BOOL LogRemoveMt(T_IPC_SRV_LOG_TYPE log_type,
                          TipcMt& mt);
```

**Remarks:** Remove a message type from a message file logging type.

**C API:** TipcSrvLogRemoveMt

### TipcSrv::StdSubjectRetrieve

**Syntax:** `static T_BOOL StdSubjectRetrieve(T_BOOL` *retrieve_time*`);`

**Remarks:** Retrieve current values for all standard subjects.

**C API:** TipcSrvStdSubjectRetrieve

### TipcSrv::StdSubjectSetSubscribe

**Syntax:** `static T_BOOL StdSubjectSetSubscribe(T_BOOL` *subscribe_flag*`,`
`T_BOOL` *time_flag*`);`

**Remarks:** Start or stop subscribing to all standard subjects.

**C API:** TipcSrvStdSubjectSetSubscribe

### TipcSrv::SubjectTraverseSubscribe

**Syntax:** `static T_PTR`
`SubjectTraverseSubscribe(T_IPC_SRV_SUBJECT_TRAV_FUNC` *func*`,`
`T_PTR` *arg*`);`

**Remarks:** Traverse the subjects to which an RTclient is subscribing.

**C API:** TipcSrvSubjectTraverseSubscribe

### TipcSrv::Stop

**Syntax:** `static T_BOOL Stop(T_IPC_SRV_STOP_TYPE` *stop_type*`);`

**Remarks:** Stop RTserver and possibly more processes.

**C API:** TipcSrvStop

## Example

This code fragment obtains a handle to RTserver by calling the static member function Instance(), creates callbacks, then uses a call to the Create member function to create the actual underlying connection to the server. It then destroys the underlying connection to RTserver.

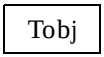
```
TipcSrv& rtserver = TipcSrv::Instance();
rtserver.CreateCbCreate(MyCallbacks::cb_server_create, NULL);

if (!rtserver) {
   // error
}

// server destroy callback
rtserver.DestroyCbCreate(MyCallbacks::cb_server_destroy, NULL);

if (!rtserver) {
 // error
}

cout << "Creating connection to RTserver." << endl;

if (!rtserver.Create(T_IPC_SRV_CONN_FULL)) {
 // error
}

// Now Destroy connection to RTserver to show how this can be done.
cout << "Destroying connection to RTserver but leave it warm."
  << endl;

if (!TipcSrv::Destroy(T_IPC_SRV_CONN_WARM)) {
 // error
}
```

# Tobj

**Name** Tobj — abstract base class for all classes in the SmartSockets C++ class library

**Synopsis** `(abstract base class)`

**Inheritance**




**Description** The Tobj class unifies error handling by managing a status flag on behalf of derived classes.

**Caution** None

**Derived Classes** TipcConn, TipcMon, TipcMonClient, TipcMonServer, TipcMsg, TipcMt

## Protected Construction/Destruction

### Tobj::Tobj

**Syntax:** `Tobj();`

**Remarks:** Create a Tobj base class.

### Tobj::~Tobj

**Syntax:** `~Tobj();`

**Remarks:** Destroy a Tobj base class.

## Public Member Functions

### Tobj::Status

**Syntax:** `T_BOOL Status() const;`

**Remarks:** Return the status of the object.

## Operators

### Tobj::operator!

**Syntax:** `T_BOOL operator!() const;`

**Remarks:** Return the negated status of the object.

## Example

This code example checks a status condition on a TipcMsg class object in two ways:

```
#include <rtworks/cxxipc.hxx>

int main()
{
  cout << "Create the message." << endl;

  TipcMsg msg(T_MT_NUMERIC_DATA);

 // FIRST way to check error flag managed in Tobj base class
  if (!msg) {
  // error
  }

 // SECOND way to check error flag managed in Tobj base class
  if (msg.Status() == FALSE) {
  // error
  }
}
```

# Chapter 7 Project Monitoring

This chapter describes how to work with the three monitoring classes of the SmartSockets C++ class library: TipcMon, TipcMonClient, and TipcMonServer.

This chapter should be considered a companion to the *TIBCO SmartSockets User's Guide*. You may wish to read the detailed information on monitoring capabilities in RTserver and RTclient processes in the *TIBCO SmartSockets User's Guide* before going further in this chapter. The intent of this chapter is to focus on creating monitoring objects from the three monitoring classes of the C++ class library.

## Topics


- *Available Monitoring Information, page 188*
- *Hierarchy Of Monitoring Classes, page 190*
- *Working With the Monitoring Classes, page 190*

# Available Monitoring Information

As described in the previous chapter, the publish-subscribe architecture of SmartSockets allows RTclient processes to easily send messages to each other, independent of where they reside on the network. In addition to enabling easy development of distributed applications, RTserver, RTmon, and RTclient processes also have monitoring capabilities and naming services. These additional services help you, as a developer (and those ultimately responsible for monitoring the deployed application), to examine detailed information about your project, as well as determining where processes are located in your network.

From within an RTclient, hundreds of pieces of information can be gathered in real time about all parts of a running SmartSockets project. These are some of the kinds of information available:

### Information about RTclients:

- names
- message buffers
- message traffic statistics
- message types
- options
- CPU usage
- memory usage
- node the RTClient is on
- options
- current time
- subjects the RTClient is subscribed to

### Information about RTservers:

- names
- message buffers
- message traffic statistics
- connections
- options
- CPU usage
- memory usage
- node the RTServer is on
- options
- current time

### Information about Subjects:

- names
- RTclients that are subscribing

### Information about Projects:

- names

This information can be either polled once to provide a one-time snapshot of information or watched to provide asynchronous updates when changes occur. A monitoring request can specify either a specific object or all objects matching a wildcard scope filter. The information is delivered to the requesting program in standard message types, prefixed with T_MT_MON_. (For brevity, the T_MT_ portion is omitted from this point on.) The fields of these monitoring message types contain the information the RTclient is interested in.

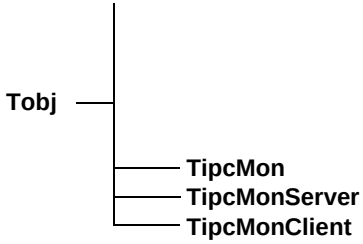
There is no monitoring available for peer-to-peer connections or for programs that do not use the SmartSockets API or C++ Class Library.

Monitoring is built into SmartSockets. No user-defined code is needed to support monitoring in an RTclient. Just as RTserver and RTclient are layered on top of the functionality of connections, monitoring is layered on top of the RTserver and RTclient architecture.

## Hierarchy Of Monitoring Classes

The three monitoring classes (TipcMon, TipcMonClient, and TipcMonServer) are derived from Tobj. The TipcMonClient class is a wrapper of the TipcMonClient* functions from the C API that take the name of an RTclient as its first argument. Likewise, the TipcMonServer class is a wrapper of the TipcMonServer* functions from the C API that take the name of an RTserver as its first argument. The third class, TipcMon, is a wrapper to the remaining TipcMon* functions that do not require the name of either an RTclient or an RTserver.

*Figure 7 Monitoring Inheritance Hierarchy*




## Working With the Monitoring Classes

This section includes a code example that shows how each of the monitoring classes can be used in an RTclient application. The source code file for this example is located in these directories:

**UNIX:**
`$RTHOME/examples/cxxipc`

**OpenVMS:**
`RTHOME:[EXAMPLES.CXXIPC]`

**Windows:**
`%RTHOME%\examples\cxxipc`

## Code Example

This RTclient creates its connection to RTserver and performs IPC monitoring. The code for the monitoring example program is:

```
 // mon.cxx -- RTclient example monitoring

#include <rtworks/cxxipc.hxx>
#define MY_CLIENT_NAME "my_client"

// ================================================================
//..mon_cb -- print monitoring msg type name
void mon_cb(T_IPC_CONN conn,
            T_IPC_CONN_PROCESS_CB_DATA data,
            T_CB_ARG arg)
{
  TipcMsg msg(data->msg);
  TipcMt  mt(msg.Type());

  TutOut("%s call yields a %s response\n", (T_STR)arg, mt.Name());
} // mon_cb

// ================================================================
//..main -- main program
int main()
{
  TipcMt mt;

  TipcSrv& server = TipcSrv::InstanceCreate(T_IPC_SRV_CONN_FULL);

  server.StdSubjectSetSubscribe(TRUE, FALSE);

  mt.Lookup(T_MT_MON_SUBJECT_SUBSCRIBE_STATUS);
  if (!mt) {
    TutOut("Message Type lookup failed.\n");
    return T_EXIT_FAILURE;
  }
  if (server.ProcessCbCreate(mt, mon_cb,
                             "mon.SubjectSubscribeWatch") == NULL)
{
    TutOut("Failed to create process callback.\n");
    return T_EXIT_FAILURE;
  }

  mt.Lookup(T_MT_MON_CLIENT_TIME_POLL_RESULT);
  if (!mt) {
    TutOut("Message Type lookup failed.\n");
    return T_EXIT_FAILURE;
  }
  if (server.ProcessCbCreate(mt, mon_cb,
                             "client_mon.TimePoll") == NULL) {
    TutOut("Failed to create process callback.\n");
    return T_EXIT_FAILURE;

  }
```

```
  mt.Lookup(T_MT_MON_SERVER_GENERAL_POLL_RESULT);
  if (!mt) {
    TutOut("Message Type lookup failed.\n");
    return T_EXIT_FAILURE;
  }
  if (server.ProcessCbCreate(mt, mon_cb,
                             "server_mon.GeneralPoll") == NULL) {
    TutOut("Failed to create process callback.\n");
    return T_EXIT_FAILURE;

  }

  TipcMon mon;
  TipcMonClient client_mon(T_IPC_MON_ALL);
  TipcMonServer server_mon(T_IPC_MON_ALL);

  if (!mon.SubjectSubscribeWatch("_all", TRUE)) {
    TutOut("Failed to setup watch.\n");
  }

  if (!client_mon.TimePoll()) {
    TutOut("Failed to poll.\n");
  }

  if (!server_mon.GeneralPoll()) {
    TutOut("Failed to poll.\n");
  }

  if (!server.MainLoop(10.0)){
    if (TutErrNumGet() != T_ERR_TIMEOUT_REACHED) {
      TutOut("Server main loop failed.\n");
    }
  }
  return T_EXIT_SUCCESS;
} // main
```

## Running a Monitoring Program

Compile and link the monitoring program (see Compiling, Linking, and Running on page 13 for generic instructions).

Start up RTserver, then use this command to run the monitoring program:

**UNIX:**

```
$ mon.x
```

**OpenVMS:**

```
$ run mon.exe
```

**Windows:**

```
$ mon.exe
```

The output from the RTclient process is similar to this:

```
Connecting to project <rtworks> on <_node> RTserver.
Using local protocol.
Message from RTserver: Connection established.
Start subscribing to subject </_workstation1_7876>.
Start subscribing to subject </_workstation1>.
Start subscribing to subject </_all>.
mon.SubjectSubscribeWatch call yields a
mon_subject_subscribe_status response
server_mon.GeneralPoll call yields a
mon_server_general_poll_result response
client_mon.TimePoll call yields a mon_client_time_poll_result
response
```

# Index

## Symbols

! (not) 186
<< (insertion) 146, 159
= (assignment) 145
>> (extraction)
  TipcMsg 32
  TipcMsgFile 39, 159

## A

abstract base class 73
access functions 75
accessing data fields from a message 110
appending data fields to a message 110
automatic flush size, getting or setting 83

## B

base class 81
block mode, getting or setting 83

## C

C++ class library
  cxxipc 1
  sscpp 1
C++ classes
  inheritance hierarchy 5
  TipcConn 95, 97, 98, 168, 185
    manpage 81
  TipcConnClient
    description 73
    manpage 95
  TipcConnServer
    description 73
    manpage 97, 98
  TipcMon 104, 108, 185
    description 73
    manpage 99, 102
  TipcMonClient 99, 108, 185
    description 73
    manpage 104, 107
  TipcMonServer 99, 104, 185
    description 73
    manpage 108, 109
  TipcMsg 161, 185
    description 73
    manpage 110
  TipcMsgFile
    description 73
    manpage 158
  TipcMt 185
    description 73
    manpage 161
  TipcSrv
    description 73
    manpage 168
  Tobj 110, 161
    description 73
    manpage 185, 186
C++ compiler
  using 77
callback 43
callback functions
  creating 184
  data structure manipulation in 75
  defined 57
case sensitivity xiv
  on UNIX and Windows xiv

CC environment variable 77, 78
compiling and linking
  examples
    invoking rtlink script on UNIX 77
connection functions
  TipcConnAccept 98
  TipcConnCheck 83
  TipcConnCreate 81, 95, 97
  TipcConnCreateClient 95
  TipcConnCreateServer 97
  TipcConnDecodeCbCreate 84
  TipcConnDecodeCbLookup 84
  TipcConnDefaultCbCreate 84
  TipcConnDefaultCbLookup 84
  TipcConnDestroy 81
  TipcConnEncodeCbCreate 85
  TipcConnEncodeCbLookup 85
  TipcConnErrorCbCreate 85
  TipcConnErrorCbLookup 85
  TipcConnFlush 86
  TipcConnGetArch 83, 85
  TipcConnGetAutoFlushSize 83
  TipcConnGetBlockMode 83
  TipcConnGetGmdMaxSize 86
  TipcConnGetGmdNumPending 87
  TipcConnGetNode 88
  TipcConnGetNumQueued 88
  TipcConnGetPid 88
  TipcConnGetSocket 91
  TipcConnGetTimeout 91
  TipcConnGetUniqueSubject 91
  TipcConnGetUser 92
  TipcConnGetXtSource 92
  TipcConnGmdFileCreate 86
  TipcConnGmdFileDelete 86
  TipcConnGmdMsgDelete 86
  TipcConnGmdMsgResend 87
  TipcConnGmdResend 87
  TipcConnKeepAlive 87
  TipcConnLock 87
  TipcConnMainLoop 88
  TipcConnMsgInsert 87
  TipcConnMsgNext 88
  TipcConnMsgProcess 88
  TipcConnMsgSearch 90
  TipcConnMsgSearchType 90
  TipcConnMsgSend 90
  TipcConnMsgSendRpc 91
  TipcConnMsgWrite 92
  TipcConnMsgWriteVa 92
  TipcConnProcessCbCreate 89
  TipcConnProcessCbLookup 89
  TipcConnQueueCbCreate 89
  TipcConnQueueCbLookup 89
  TipcConnRead 89
  TipcConnReadCbCreate 90
  TipcConnReadCbLookup 90
  TipcConnSetAutoFlushSize 83
  TipcConnSetBlockMode 83
  TipcConnSetGmdMaxSize 86
  TipcConnSetSocket 91
  TipcConnSetTimeout 91
  TipcConnUnlock 91
  TipcConnWriteCbCreate 92
  TipcConnWriteCbLookup 92
connections
  accepting client 56
  callback 57
  class hierarchy 42
  creating 54
    client side 55, 95
    server side 54, 184
  defined 41
  destroying 56, 184
  dummy 43
  example 43
constructing message types 161
constructing messages 110
conventions used in this manual xi
creating a client connection 95
creating a server connection 97
customer support xiv
-cxx flag 78
cxxipc library 1

## D

data fields
  accessing 110
  appending 110
definitions
  connection 41
  message 23
  message type 23
descriptions
  TipcConnClient C++ class 73
  TipcConnServer C++ class 73
  TipcMon C++ class 73
  TipcMonClient C++ class 73
  TipcMonServer C++ class 73
  TipcMsg C++ class 73
  TipcMsgFile C++ class 73
  TipcMt C++ class 73
  TipcSrv C++ class 73
  Tobj C++ class 73
destroy status
  specifying 168
destroying message types 161
destroying messages 110
destroying RTserver connections 184
dummy connections 81
dynamic message routing 60

## E

error handling 11, 79, 185
  and static member functions 18
error number 79
examples
  calling a static member function 184
  checking a status condition on a TipcMsg class object 186
  constructing a NUMERIC_DATA message 156
  creating a client connection 96
  creating a file stream 159
  creating a message type 167
  creating a TCP server connection 98
  creating server connections 184
  destroying an RTserver connection 184
  naming an executable, on OpenVMS 78
  setting CC from a makefile 78
  using the GNU C++ compiler 78
extraction operator, using 32

## F

file names
  specifying xiv
file stream 159
flush size, getting or setting 83
functions
  case-sensitivity xiv
  connection
    TipcConnAccept 98
    TipcConnCheck 83
    TipcConnCreate 81, 95, 97
    TipcConnCreateClient 95
    TipcConnCreateServer 97
    TipcConnDecodeCbCreate 84
    TipcConnDecodeCbLookup 84
    TipcConnDefaultCbCreate 84
    TipcConnDefaultCbLookup 84
    TipcConnDestroy 81
    TipcConnEncodeCbCreate 85
    TipcConnEncodeCbLookup 85
    TipcConnErrorCbCreate 85
    TipcConnErrorCbLookup 85
    TipcConnFlush 86
    TipcConnGetArch 83, 85
    TipcConnGetAutoFlushSize 83
    TipcConnGetBlockMode 83
    TipcConnGetGmdMaxSize 86
    TipcConnGetGmdNumPending 87
    TipcConnGetNode 88
    TipcConnGetNumQueued 88
    TipcConnGetPid 88
    TipcConnGetSocket 91
    TipcConnGetTimeout 91
    TipcConnGetUniqueSubject 91
    TipcConnGetUser 92
    TipcConnGetXtSource 92

TipcConnGmdFileCreate 86
TipcConnGmdFileDelete 86
TipcConnGmdMsgDelete 86
TipcConnGmdMsgResend 87
TipcConnGmdResend 87
TipcConnKeepAlive 87
TipcConnLock 87
TipcConnMainLoop 88
TipcConnMsgInsert 87
TipcConnMsgNext 88
TipcConnMsgProcess 88
TipcConnMsgSearch 90
TipcConnMsgSearchType 90
TipcConnMsgSend 90
TipcConnMsgSendRpc 91
TipcConnMsgWrite 92
TipcConnMsgWriteVa 92
TipcConnProcessCbCreate 89
TipcConnProcessCbLookup 89
TipcConnQueueCbCreate 89
TipcConnQueueCbLookup 89
TipcConnRead 89
TipcConnReadCbCreate 90
TipcConnReadCbLookup 90
TipcConnSetAutoFlushSize 83
TipcConnSetBlockMode 83
TipcConnSetGmdMaxSize 86
TipcConnSetSocket 91
TipcConnSetTimeout 91
TipcConnUnlock 91
TipcConnWriteCbCreate 92
TipcConnWriteCbLookup 92
IPC
TipcGetGmdDir 93
message
TipcMsg* 110
TipcMsgAddNamedBinary 112, 113
TipcMsgAddNamedBinaryPtr 112, 115
TipcMsgAddNamedChar 113
TipcMsgAddNamedInt2 113
TipcMsgAddNamedInt2Array 113
TipcMsgAddNamedInt2ArrayPtr 114
TipcMsgAddNamedInt4 114
TipcMsgAddNamedInt4Array 113, 114
TipcMsgAddNamedInt4ArrayPtr 114
TipcMsgAddNamedInt8 114
TipcMsgAddNamedInt8Array 114
TipcMsgAddNamedInt8ArrayPtr 113, 114
TipcMsgAddNamedMsg 115
TipcMsgAddNamedMsgArray 115
TipcMsgAddNamedMsgArrayPtr 115
TipcMsgAddNamedReal16 116
TipcMsgAddNamedReal16Array 116
TipcMsgAddNamedReal16ArrayPtr 116
TipcMsgAddNamedReal4 115
TipcMsgAddNamedReal4ArrayPtr 116
TipcMsgAddNamedReal8 116, 117
TipcMsgAddNamedReal8Array 116, 117
TipcMsgAddNamedReal8ArrayPtr 116, 118
TipcMsgAddNamedStr 117
TipcMsgAddNamedStrArray 117
TipcMsgAddNamedStrArrayPtr 117
TipcMsgAddNamedStrPtr 117, 118
TipcMsgAddNamedUnknown 117
TipcMsgAddNamedXml 118
TipcMsgAddNamedXmlPtr 118
TipcMsgAppend* 110
TipcMsgAppendMsgArray 120
TipcMsgDeleteCurrent 126
TipcMsgDeleteField 126
TipcMsgDeleteNamedField 126
TipcMsgGetNamed 128, 130
TipcMsgGetNamedBinary 128
TipcMsgGetNamedBool 128
TipcMsgGetNamedBoolArray 128
TipcMsgGetNamedByte 128
TipcMsgGetNamedChar 128
TipcMsgGetNamedInt2 128
TipcMsgGetNamedInt2Array 128
TipcMsgGetNamedInt4 129
TipcMsgGetNamedInt4Array 129
TipcMsgGetNamedInt8 129
TipcMsgGetNamedInt8Array 129
TipcMsgGetNamedMsg 129
TipcMsgGetNamedMsgArray 129
TipcMsgGetNamedReal16 130
TipcMsgGetNamedReal16Array 130
TipcMsgGetNamedReal4 129
TipcMsgGetNamedReal4Array 130
TipcMsgGetNamedReal8 130

TipcMsgGetNamedReal8Array 130
TipcMsgGetNamedStr 130
TipcMsgGetNamedStrArray 130
TipcMsgGetNamedUnknown 131
TipcMsgGetNamedUtf8 131
TipcMsgGetNamedUtf8Array 131
TipcMsgGetNamedXml 131
TipcMsgGetType 75
TipcMsgIncrRefCount 131
TipcMsgNext* 110
TipcMsgSetType 75
TipcMsgUpdateNamedBinary 139
TipcMsgUpdateNamedBinaryPtr 139
TipcMsgUpdateNamedBool 140
TipcMsgUpdateNamedBoolArray 140
TipcMsgUpdateNamedBoolArrayPtr 140
TipcMsgUpdateNamedByte 140
TipcMsgUpdateNamedChar 140
TipcMsgUpdateNamedInt2 140
TipcMsgUpdateNamedInt2Array 140
TipcMsgUpdateNamedInt2ArrayPtr 140
TipcMsgUpdateNamedInt4 141
TipcMsgUpdateNamedInt4Array 141
TipcMsgUpdateNamedInt4ArrayPtr 141
TipcMsgUpdateNamedInt8 141
TipcMsgUpdateNamedInt8Array 141
TipcMsgUpdateNamedInt8ArrayPtr 141
TipcMsgUpdateNamedMsg 141
TipcMsgUpdateNamedMsgArray 141
TipcMsgUpdateNamedMsgArrayPtr 142
TipcMsgUpdateNamedReal16 143
TipcMsgUpdateNamedReal16Array 143
TipcMsgUpdateNamedReal16ArrayPtr 143
TipcMsgUpdateNamedReal4 142
TipcMsgUpdateNamedReal4Array 142
TipcMsgUpdateNamedReal4ArrayPtr 142
TipcMsgUpdateNamedReal8 142
TipcMsgUpdateNamedReal8Array 142
TipcMsgUpdateNamedReal8ArrayPtr 142
TipcMsgUpdateNamedStr 143
TipcMsgUpdateNamedStrArray 143
TipcMsgUpdateNamedStrArrayPtr 143
TipcMsgUpdateNamedStrPtr 143
TipcMsgUpdateNamedUtf8 143
TipcMsgUpdateNamedUtf8ArrayPtr 144
TipcMsgUpdateNamedUtf8Ptr 144
TipcMsgUpdateNamedUtfArray 144
TipcMsgUpdateNamedXml 144
TipcMsgUpdateNamedXmlPtr 144
message type
TipcMtLookup 74
monitoring
TipcMonClientNamesGetWatch 100
TipcMonClientNamesPoll 100
RTserver communication
TipcSrvMsgSend 75
utility 77
TutErrNumGet 79
TutErrNumSet 79

## G

guaranteed message delivery (GMD) 60, 171

## I

identifiers
case sensitivity xiv
include files 18, 30, 32, 43, 63, 190
inheritance 6, 8
inheritance hierarchy 5
insertion operator, using 32
IOstream
classes 110
syntax, using for SmartSockets messages 158
IPC functions
TipcGetGmdDir 93

## L

load balancing 60
looking up message types 161

## M

macros, utility 77
make utility, compiling with 78
manipulating messages 110
message
  accessing data fields 13
  appending data fields 11
  array data fields 34
  building in C++ 25, 27
message file
  creating 38
  opening 38
  reading 39
message functions
  TipcMsg* 110
  TipcMsgAddNamedArrayPtr 116, 118
  TipcMsgAddNamedBinary 112, 113
  TipcMsgAddNamedBinaryPtr 112, 115
  TipcMsgAddNamedChar 113
  TipcMsgAddNamedInt2 113
  TipcMsgAddNamedInt2Array 113
  TipcMsgAddNamedInt2ArrayPtr 114
  TipcMsgAddNamedInt4 114
  TipcMsgAddNamedInt4Array 113, 114
  TipcMsgAddNamedInt4ArrayPtr 114
  TipcMsgAddNamedInt8 114
  TipcMsgAddNamedInt8Array 114
  TipcMsgAddNamedInt8ArrayPtr 113, 114
  TipcMsgAddNamedMsg 115
  TipcMsgAddNamedMsgArray 115
  TipcMsgAddNamedMsgArrayPtr 115
  TipcMsgAddNamedReal16 116
  TipcMsgAddNamedReal16Array 116
  TipcMsgAddNamedReal16ArrayPtr 116
  TipcMsgAddNamedReal4 115
  TipcMsgAddNamedReal4ArrayPtr 116
  TipcMsgAddNamedReal8 116, 117
  TipcMsgAddNamedReal8Array 116, 117
  TipcMsgAddNamedStr 117
  TipcMsgAddNamedStrArray 117
  TipcMsgAddNamedStrArrayPtr 117
  TipcMsgAddNamedStrPtr 117, 118
  TipcMsgAddNamedUnknown 117
  TipcMsgAddNamedXml 118
  TipcMsgAddNamedXmlPtr 118
  TipcMsgAppend* 110
  TipcMsgAppendMsgArray 120
  TipcMsgDeleteCurrent 126
  TipcMsgDeleteField 126
  TipcMsgDeleteNamedField 126
  TipcMsgGetNamed 128, 130
  TipcMsgGetNamedBinary 128
  TipcMsgGetNamedBool 128
  TipcMsgGetNamedBoolArray 128
  TipcMsgGetNamedByte 128
  TipcMsgGetNamedChar 128
  TipcMsgGetNamedInt2 128
  TipcMsgGetNamedInt2Array 128
  TipcMsgGetNamedInt4 129
  TipcMsgGetNamedInt4Array 129
  TipcMsgGetNamedInt8 129
  TipcMsgGetNamedInt8Array 129
  TipcMsgGetNamedMsg 129
  TipcMsgGetNamedMsgArray 129
  TipcMsgGetNamedReal16 130
  TipcMsgGetNamedReal16Array 130
  TipcMsgGetNamedReal4 129
  TipcMsgGetNamedReal4Array 130
  TipcMsgGetNamedReal8 130
  TipcMsgGetNamedReal8Array 130
  TipcMsgGetNamedStr 130
  TipcMsgGetNamedStrArray 130
  TipcMsgGetNamedUnknown 131
  TipcMsgGetNamedUtf8 131
  TipcMsgGetNamedUtf8Array 131
  TipcMsgGetNamedXml 131
  TipcMsgGetType 75
  TipcMsgIncrRefCount 131
  TipcMsgNext* 110
  TipcMsgNextInt4Array 133
  TipcMsgNextInt8Array 134
  TipcMsgSetType 75
  TipcMsgUpdateNamedBinary 139
  TipcMsgUpdateNamedBinaryPtr 139
  TipcMsgUpdateNamedBool 140
  TipcMsgUpdateNamedBoolArray 140
  TipcMsgUpdateNamedBoolArrayPtr 140
  TipcMsgUpdateNamedByte 140
  TipcMsgUpdateNamedChar 140

TipcMsgUpdateNamedInt2 140
TipcMsgUpdateNamedInt2Array 140
TipcMsgUpdateNamedInt2ArrayPtr 140
TipcMsgUpdateNamedInt4 141
TipcMsgUpdateNamedInt4Array 141
TipcMsgUpdateNamedInt4ArrayPtr 141
TipcMsgUpdateNamedInt8 141
TipcMsgUpdateNamedInt8Array 141
TipcMsgUpdateNamedInt8ArrayPtr 141
TipcMsgUpdateNamedMsg 141
TipcMsgUpdateNamedMsgArray 141
TipcMsgUpdateNamedMsgArrayPtr 142
TipcMsgUpdateNamedReal16 143
TipcMsgUpdateNamedReal16Array 143
TipcMsgUpdateNamedReal16ArrayPtr 143
TipcMsgUpdateNamedReal4 142
TipcMsgUpdateNamedReal4Array 142
TipcMsgUpdateNamedReal4ArrayPtr 142
TipcMsgUpdateNamedReal8 142
TipcMsgUpdateNamedReal8Array 142
TipcMsgUpdateNamedReal8ArrayPtr 142
TipcMsgUpdateNamedStr 143
TipcMsgUpdateNamedStrArray 143
TipcMsgUpdateNamedStrArrayPtr 143
TipcMsgUpdateNamedStrPtr 143
TipcMsgUpdateNamedUtf8 143
TipcMsgUpdateNamedUtf8ArrayPtr 144
TipcMsgUpdateNamedUtf8Ptr 144
TipcMsgUpdateNamedXml 144
TipcMsgUpdateNamedXmlPtr 144
message type functions
TipcMtLookup 74
message types
constructing and destroying 161
defined 23
standard or user-defined 26
messages
case sensitivity xiv
constructing and destroying 110
defined 23
monitoring
client information 190
general RTclients 99
information available 188
server information 190
specific RTclients 104
specific RTserver 108
monitoring functions
TipcMonClientNamesGetWatch 100
TipcMonClientNamesPoll 100
multiple connections
the sscpp library 7

## N

namespace 60
native C++ compiler, using 77

## O

opaque data type 75
options
case sensitivity xiv

## P

problem flag 151
publish-subscribe model 59

## R

reference counts 37
RTHOME directory
location on OpenVMS and Windows 76
rtlink -cxx flag 77
rtlink shell script 77
rtlink syntax on OpenVMS 78
RTserver and RTclient functionality 60
RTserver communication functions
TipcSrvMsgSend 75
rtserver64 command 14

## S

server connections
  creating and destroying 184
shell commands
  specifying xiv
SmartSockets C API 3
  how to find matching member functions 7
  mapping to C++ classes 7
  object-oriented aspects 5
SmartSockets C++ Class Library
  advantages 8
  an example 10
  mapping to C API 7
  object-oriented aspects 5
  relation to C API 3
  source code 19
  using the index 7
  when to use 8
source code 19
specifying a destroy status 168
sscpp library 1
starting an RTserver 60
static member functions
  error handling 18
status flag 185
  checking, in TObj class 79
structures
  T_IPC_CONN
    in TipcConn 42
  T_IPC_MSG
    in TipcMsg 24
  T_IPC_MT
    constructing TipcMt objects 26
    in TipcMt 26
support, contacting xiv

## T

technical support xiv
TipcConn 7, 20, 43
TipcConn C++ class 95, 97, 98, 168, 185
  manpage 81
TipcConnAccept connection function 62, 98
TipcConnCheck connection function 83
TipcConnClient 7, 20, 62, 81
TipcConnClient C++ class
  description 73
  manpage 95
TipcConnCreate connection function 43, 81, 95, 97
TipcConnCreateClient connection function 73, 95
TipcConnCreateServer connection function 73, 97
TipcConnDecodeCbCreate connection function 84
TipcConnDecodeCbLookup connection function 84
TipcConnDefaultCbCreate connection function 84
TipcConnDefaultCbLookup connection function 84
TipcConnDestroy connection function 81
TipcConnEncodeCbCreate connection function 85
TipcConnEncodeCbLookup connection function 85
TipcConnErrorCbCreate connection function 85
TipcConnErrorCbLookup connection function 85
TipcConnFlush connection function 86
TipcConnGetArch connection function 83, 85
TipcConnGetAutoFlushSize connection function 83
TipcConnGetBlockMode connection function 83
TipcConnGetGmdMaxSize connection function 86
TipcConnGetGmdNumPending connection function 87
TipcConnGetNode connection function 88
TipcConnGetNumQueued connection function 88
TipcConnGetPid connection function 88
TipcConnGetSocket connection function 91
TipcConnGetTimeout connection function 91
TipcConnGetUniqueSubject connection function 91
TipcConnGetUser connection function 92
TipcConnGetXtSource connection function 92
TipcConnGmdFileCreate connection function 86
TipcConnGmdFileDelete connection function 86
TipcConnGmdMsgDelete connection function 86
TipcConnGmdMsgResend connection function 87
TipcConnGmdResend connection function 87
TipcConnKeepAlive connection function 87
TipcConnLock connection function 87
TipcConnMainLoop connection function 88
TipcConnMsgInsert connection function 87
TipcConnMsgNext connection function 88
TipcConnMsgProcess connection function 88
TipcConnMsgSearch connection function 90

TipcConnMsgSearchType connection function 90
TipcConnMsgSend connection function 90
TipcConnMsgSendRpc connection function 91
TipcConnMsgWrite connection function 92
TipcConnMsgWriteVa connection function 92
TipcConnProcessCbCreate connection function 89
TipcConnProcessCbLookup connection function 89
TipcConnQueueCbCreate connection function 89
TipcConnQueueCbLookup connection function 89
TipcConnRead connection function 89
TipcConnReadCbCreate connection function 90
TipcConnReadCbLookup connection function 90
TipcConnServer 20
TipcConnServer C++ class
  description 73
  manpage 97, 98
TipcConnSetAutoFlushSize connection function 83
TipcConnSetBlockMode connection function 83
TipcConnSetGmdMaxSize connection function 86
TipcConnSetSocket connection function 91
TipcConnSetTimeout connection function 91
TipcConnUnlock connection function 91
TipcConnWriteCbCreate connection function 92
TipcConnWriteCbLookup connection function 92
TipcGetGmdDir IPC function 93
TipcLocalClient 7
TipcMon 7, 20
  an example 190
TipcMon C++ class 104, 108, 185
  description 73
  manpage 99, 102
TipcMonClient 7, 20, 190
  an example 190
TipcMonClient C++ class 99, 108, 185
  description 73
  manpage 104, 107
TipcMonClientNamesGetWatch monitoring function 100
TipcMonClientNamesPoll monitoring function 100
TipcMonServer 7, 20, 190
  an example 190
TipcMonServer C++ class 99, 104, 185
  description 73
  manpage 108, 109
TipcMsg 7, 20
  accessing data fields 13, 30, 32
  Append() 30
    an example 30
  appending data fields 11, 30, 32
  array data fields 34
  construction 25, 27
    using a message type name 29
    using a message type number 28
    using a simple constructor 28
    using a T_IPC_MSG variable 29
    using a TipcMsg object 29
    using a TipcMt object 28
    using the convenience constructor 28
    vacant object 28
  copy constructor 29
  error checking 11
  extraction operator 32
  insertion and extraction operators 32
  insertion operator 32
  manipulators 20
    calling order 35
    Check 11
    Check() 34
    GetSize() 34
    SetSize() 34
  Next() 30
    an example 30
  non-vacant objects 25
  reference counts 37
  vacant objects 25, 28
TipcMsg C++ class 161, 185
  description 73
  manpage 110
TipcMsg* message function 110
TipcMsgAddNamedBinary message function 112, 113
TipcMsgAddNamedBinaryPtr message function 112, 115
TipcMsgAddNamedChar message function 113
TipcMsgAddNamedInt2 message function 113
TipcMsgAddNamedInt2Array message function 113
TipcMsgAddNamedInt2ArrayPtr message function 114
TipcMsgAddNamedInt4 message function 114

TipcMsgAddNamedInt4Array message function 113, 114
TipcMsgAddNamedInt4ArrayPtr message function 114
TipcMsgAddNamedInt8 message function 114
TipcMsgAddNamedInt8Array message function 114
TipcMsgAddNamedInt8ArrayPtr message function 113, 114
TipcMsgAddNamedMsg message function 115
TipcMsgAddNamedMsgArray message function 115
TipcMsgAddNamedMsgArrayPtr message function 115
TipcMsgAddNamedReal16 message function 116
TipcMsgAddNamedReal16Array message function 116
TipcMsgAddNamedReal16ArrayPtr message function 116
TipcMsgAddNamedReal4 message function 115
TipcMsgAddNamedReal4ArrayPtr message function 116
TipcMsgAddNamedReal8 message function 116, 117
TipcMsgAddNamedReal8Array message function 116, 117
TipcMsgAddNamedReal8ArrayPtr message function 116, 118
TipcMsgAddNamedStr message function 117
TipcMsgAddNamedStrArray message function 117
TipcMsgAddNamedStrArrayPtr message function 117
TipcMsgAddNamedStrPtr message function 117, 118
TipcMsgAddNamedUnknown message function 117
TipcMsgAddNamedXml message function 118
TipcMsgAddNamedXmlPtr message function 118
TipcMsgAppend* message function 110
TipcMsgAppendMsgArray message function 35, 120
TipcMsgDeleteCurrent message function 126
TipcMsgDeleteField message function 126
TipcMsgDeleteNamedField message function 126
TipcMsgFile 20, 38
TipcMsgFile C++ class
  description 73
  manpage 158
TipcMsgFileWrite message function 73
TipcMsgGetNamed Binary message function 128
TipcMsgGetNamed message function 128, 130
TipcMsgGetNamedBool message function 128
TipcMsgGetNamedBoolArray message function 128
TipcMsgGetNamedByte message function 128
TipcMsgGetNamedChar message function 128
TipcMsgGetNamedInt2 message function 128
TipcMsgGetNamedInt2Array message function 128
TipcMsgGetNamedInt4 message function 129
TipcMsgGetNamedInt4Array message function 129
TipcMsgGetNamedInt8 message function 129
TipcMsgGetNamedInt8Array message function 129
TipcMsgGetNamedMsg message function 129
TipcMsgGetNamedMsgArray message function 129
TipcMsgGetNamedReal16 message function 130
TipcMsgGetNamedReal16Array message function 130
TipcMsgGetNamedReal4 message function 129
TipcMsgGetNamedReal4Array message function 130
TipcMsgGetNamedReal8 message function 130
TipcMsgGetNamedReal8Array message function 130
TipcMsgGetNamedStr message function 130
TipcMsgGetNamedStrArray message function 130
TipcMsgGetNamedUnknown message function 131
TipcMsgGetNamedUtf8 message function 131
TipcMsgGetNamedUtf8Array message function 131
TipcMsgGetNamedXml message function 131
TipcMsgGetType message function 75
TipcMsgIncrRefCount message function 131
TipcMsgNext* message function 110
TipcMsgNextInt4Array message function 133
TipcMsgNextInt8Array message function 134
TipcMsgNextMsgArray message function 35
TipcMsgSetType message function 75
TipcMsgUpdateNamedBinary message function 139
TipcMsgUpdateNamedBinaryPtr message function 139
TipcMsgUpdateNamedBool message function 140
TipcMsgUpdateNamedBoolArray message function 140
TipcMsgUpdateNamedBoolArrayPtr message function 140
TipcMsgUpdateNamedByte message function 140
TipcMsgUpdateNamedChar message function 140
TipcMsgUpdateNamedInt2 message function 140
TipcMsgUpdateNamedInt2Array message function 140

TipcMsgUpdateNamedInt2ArrayPtr message function 140
TipcMsgUpdateNamedInt4 message function 141
TipcMsgUpdateNamedInt4Array message function 141
TipcMsgUpdateNamedInt4ArrayPtr message function 141
TipcMsgUpdateNamedInt8 message function 141
TipcMsgUpdateNamedInt8Array message function 141
TipcMsgUpdateNamedInt8ArrayPtr message function 141
TipcMsgUpdateNamedMsg message function 141
TipcMsgUpdateNamedMsgArray message function 141
TipcMsgUpdateNamedMsgArrayPtr message function 142
TipcMsgUpdateNamedReal16 message function 143
TipcMsgUpdateNamedReal16Array message function 143
TipcMsgUpdateNamedReal16ArrayPtr message function 143
TipcMsgUpdateNamedReal4 message function 142
TipcMsgUpdateNamedReal4Array message function 142
TipcMsgUpdateNamedReal4ArrayPtr message function 142
TipcMsgUpdateNamedReal8 message function 142
TipcMsgUpdateNamedReal8Array message function 142
TipcMsgUpdateNamedReal8ArrayPtr message function 142
TipcMsgUpdateNamedStr message function 143
TipcMsgUpdateNamedStrArray message function 143
TipcMsgUpdateNamedStrArrayPtr message function 143
TipcMsgUpdateNamedStrPtr message function 143
TipcMsgUpdateNamedUtf8 message function 143
TipcMsgUpdateNamedUtf8Array message function 144
TipcMsgUpdateNamedUtf8ArrayPtr message function 144
TipcMsgUpdateNamedUtf8Ptr message function 144
TipcMsgUpdateNamedXml message function 144
TipcMsgUpdateNamedXmlPtr message function 144
TipcMt 7, 21
  construction 26
  standard message types 26
  user-defined message types 26
TipcMt C++ class 185
  description 73
  manpage 161
TipcMtLookup message type function 74
TipcSrv 7, 21, 62
  an example 63
  getting an instance of 11, 62
  InstanceCreate() 11
  SubjectSubscribe() 13
  subscribing to a subject 13
TipcSrv C++ class
  description 73
  manpage 168
TipcSrvCreate RTserver communication function 62
TipcSrvMsgSend RTserver communication function 75
Tobj 21
Tobj C++ class 110, 161
  description 73
  manpage 185, 186
TutErrNumGet function 79
TutErrNumGet utility function 79
TutErrNumSet function 79
TutErrNumSet utility function 79
TutErrNumToStr function 79
TutErrStrGet function 79

## U

user class 73
utility functions
  TutErrNumGet 79
  TutErrNumSet 79

## V

vacant objects
  TipcMsg 25, 28
  TipcMt 27
virtual member functions 6, 8

## W

wrappers 71